December 2021

Resolving Soft Error Susceptibilities Within Lossy Compressed HPC Data

Dakota Kent Fulp
Clemson University, dakotafulp@gmail.com

Resolving Soft Error Susceptibilities Within Lossy Compressed HPC Data

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Dakota Fulp
December 2021

Accepted by:
Dr. Jon C. Calhoun, Committee Chair
Dr. Melissa Smith
Dr. Walt Ligon
Dr. Rong Ge

Abstract

Due to improvements in high-performance computing (HPC) systems, researchers have created powerful applications capable of solving previously intractable problems. While solving these problems, such applications create vast amounts of data, which stresses the I/O subsystem. Researchers use lossy compression to remedy this issue by reducing the data's size, but, as we demonstrate in this thesis, a single soft error leaves lossy compressed data unusable. Due to the high information content per bit ratio, lossy compressed data is sensitive to soft errors, which is an issue as soft errors have become commonplace on HPC systems. Yet, few works have sought to resolve this significant weakness.

This thesis addresses the lack of works by performing an extensive soft error assessment and providing an approach to resolving the soft error sensitivity demonstrated by lossy compressed data. Upon evaluating the SZ and ZFP lossy compression algorithms, we find 95.28% of all trials led to error propagation and silent data corruption (SDC). Furthermore, 100% of trials using ZFP led to the same conclusion. Our findings also indicate that, on average, a single soft error propagates to ∼10% of data values. We find this trend exists for both SZ and ZFP and fluctuates with different compression ratios. Lastly, we find significant drops in the resulting data integrity due to a single soft error. Our findings indicate that all error bounding modes we test are susceptible to soft errors. The only exception is the block-based compression algorithm, which prevents the error from propagating outside the block.

Leveraging our findings, we develop ARC (Automated Resiliency for Compression). ARC automatically determines and applies the optimal error-correcting code (ECC) configuration to data while respecting user constraints on storage, throughput, and resiliency. ARC's design centers around four main goals: scalability, performance, resiliency, and ease of use. We evaluate ARC using these four goals. Upon assessing the scalability of ARC's underlying ECC algorithms, we find

each approach scales near linearly with encoding throughputs ranging from 0.04 – 3730 MB/s and decoding throughputs ranging from 10.64 – 3602 MB/s when working on a 40 core node. When evaluating how ARC satisfies user constraints, we find ARC adequately meets user needs whether they synergize or conflict with one another. After evaluating ARC's resiliency, we find ARC effectively resolves both single-bit and multi-bit soft errors depending on the provided user constraints. Lastly, we demonstrate the four lines of code needed to implement ARC and show how users should consider the failure rate of a system when choosing constraints to illustrate its ease of use. Overall, this thesis demonstrates the soft error vulnerabilities of lossy compressed data along with a practical approach to resolving these vulnerabilities.

Acknowledgments

To begin, I am incredibly grateful to my supervisor and committee chair, Dr. Jon C. Calhoun, for his guidance, support, and advice throughout my time at Clemson. I would also like to thank Dr. Robert Underwood for his invaluable assistance and patience throughout the process of completing this work. Next, I would like to thank both Dr. William M. Jones Jr. and Dr. Nathan DeBardeleben for their guidance and the opportunities they gave me that led me to Clemson University. I would also like to thank both my biological and Ekklesia church families for being a continuous source of encouragement and support throughout all of my higher education. Out of my entire family, I would like to thank my loving wife, Megan Hickman Fulp, the most, as, without her support, none of this would be possible. Finally, I am grateful to Dr. Melissa Smith, Dr. Walt Ligon, and Dr. Rong Ge for agreeing to serve as part of my committee.

The material in this thesis is based upon work supported by the National Science Foundation under Grant No. SHF-1910197, SHF-1943114, MRI-#1725573, and NRT-DESE 1633608. The material in this thesis is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE) for the DOE. ORISE is managed by ORAU under contract number DE-SC0014664. All opinions expressed in this thesis are the authors and do not necessarily reflect the policies and views of DOE, ORAU, or ORISE. Clemson University is acknowledged for generous allotment of computing time on the Palmetto cluster.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past 60 years, there have been significant advancements in high-performance computing (HPC) capabilities, leading to the creation of many cutting-edge systems. Each system makes various improvements upon the previous, such as providing further resources, like extra processors and memory, for researchers to leverage. The addition of these resources leads to more floating-point operations per second (FLOPS), which enable scientists to undertake previously intractable problems. For instance, due to the extra resources and improved performance of newer systems, ground-breaking simulations such as the Hurricane Isabel [1], Nyx dark matter [2], and Community Earth System Model (CESM) [18] simulations were made possible. However, these simulations also generate significant quantities of data per run. In particular, the Hurricane Isabel simulation produces 48 1.25 GB snapshots per run. Furthermore, the Nyx and CESM simulations generate hundreds of snapshots per run, leading to more than 20 PB of data generated in each case. Even though HPC systems can efficiently handle the computations needed by these simulations, the immense volumes of data each simulation creates only worsen the current I/O bottleneck found in HPC systems.

While HPC system capabilities have improved considerably, not all areas have seen the same amount of improvement. Cappello et al.'s 2019 study demonstrates this disparity by comparing four leading HPC systems that were online between 2008 and 2018 [9], as seen in Table 1.1. Among these four systems, the authors find significant gains in FLOPS and memory size with improvements of 114× and 27.8× over the ten years, respectively. However, the I/O bandwidth only achieves a 10.4× improvement over the same period, and as this disparity grows, issues arise in the system.

HPC System	Year	PetaFLOPS	Memory	I/O
CRAY Jaguar	2008	1.75	360 TB	240 GB/s
CRAY Blue Waters	2012	13.3	1.5 PB	1.1 TB/s
CRAY CORI	2017	30	1.4 PB	1.7 TB/s
IBM Summit	2018	200	>10 PB	2.5 TB/s

Table 1.1: Performance of leading HPC systems from 2008 to 2018, Courtesy of [9].

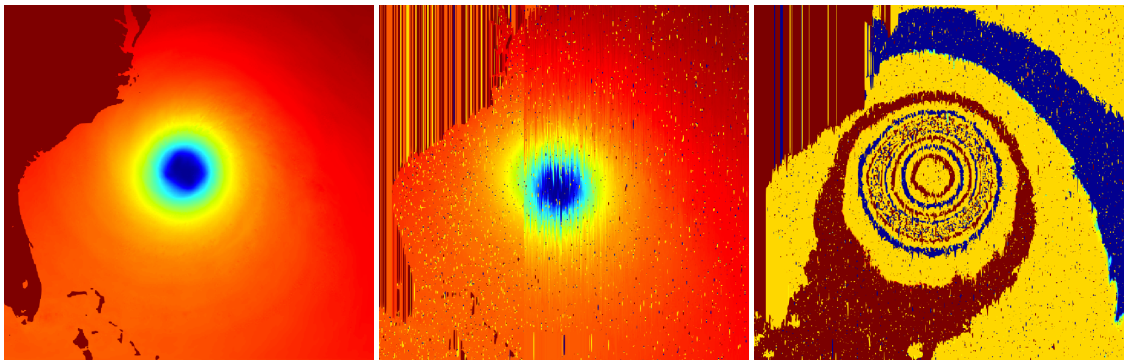
Lossless Scheme	Compression Ratio
FPC	$1.02\times \sim 1.96\times$
ISOBAR	$1.12\times \sim 1.48\times$
PRIMACY	$1.13\times \sim 2.16\times$
ALACRITY	$1.19\times \sim 1.58\times$
CC	up to $2.13\times$
IOFSL	$\sim 1.9\times$
Binary Masking	$1.11\times \sim 1.33\times$
MCRENGINE	up to $1.18\times$

Table 1.2: Floating-point performance of leading lossless compression schemes, Courtesy of [34].

On the one hand, as the gap between FLOPS and I/O bandwidth increases, the system spends more time moving data instead of performing meaningful computations. On the other hand, as the gap between the memory size and I/O bandwidth grows, the system requires more time to store all data in memory within the file system. In both cases, these system issues result in processor underutilization and limited application efficiency.

To resolve I/O issues that arise from large datasets, researchers often leverage data reduction methods such as lossless or lossy compression to reduce the data's size. Lossless compression algorithms, such as ZStd [10] or GZip [15], reduce data size with no loss in decompressed data precision. By using Huffman and Arithmetic encoding, these algorithms remove redundant bit sequences in the data. However, due to the high entropy found in the mantissa bits of the IEEE 754 floating-point standard most scientific datasets use, these algorithms suffer from low compression ratios when used on scientific data. Specifically, Son et al. find that leading lossless compression algorithms only achieve compression ratios just above $2\times$ when working with floating-point data [34]; we provide these results in Table 1.2.

To achieve higher compression ratios on scientific datasets, researchers utilize lossy compression algorithms such as SZ [12, 39, 23] and ZFP [25]. Lossy compression algorithms operate under the assumption that full data precision is not always necessary. By reducing data precision, these algorithms do not suffer from the high entropy of the mantissa bits and result in higher compression



(a) Control (b) 49.6% Incorrect (c) 99.4% Incorrect

Figure 1.1: Effect of a soft error at two different locations in the lossy compressed Hurricane dataset.

ratios than lossless compression algorithms. As introducing any error into the data must be carefully controlled, lossy compression algorithms allow users to determine the acceptable amount of error they introduce using error-bounding modes and values.

When working with HPC applications and data, there is always a possibility of encountering soft errors. Therefore, it is critical to understand the resiliency of all applications and data, including any changes to either. Using Sridharan et al.'s findings [37, 35], we determine in Chapter 5 that soft error failures occur on average every 1.9 days on the Cielo HPC system, which does not include undetected soft errors that cause silent data corruption (SDC). To demonstrate the effects of soft errors on lossy compressed data, we manually inject a single-bit soft error in two different bit locations within the Hurricane Isabel dataset compressed using the SZ lossy compression algorithm and an absolute error bound of 0.1. In Figure 1.1(b), the soft error impacts bit 400,005 of the compressed data, while in Figure 1.1(c), the soft error impacts bit 465,840. In both cases, the decompressed dataset demonstrates a high percent of incorrect elements, defined as the number of data values whose error violates the target user-defined error bound. Particularly, Figure 1.1(b) and Figure 1.1(c) have 49.6% and 99.4% incorrect elements, respectively. Even though this sensitivity to soft errors is critical, few works have aimed to understand the impact on and protect lossy compressed data from soft errors [21, 22, 27, 32].

This research contributes a better understanding of bit corruption's impact on lossy compressed data along with an approach to protecting lossy compressed data from soft errors. First, we conduct an extensive fault injection study using two industry-standard lossy compressors, SZ [12, 39, 23] and ZFP [25]. Using our findings as a guide, we develop ARC (Automated Resiliency for

Compression) to protect lossy compressed data fidelity from soft errors. ARC automatically defends lossy compressed data using the optimal error-correcting codes (ECC) configuration while abiding by user constraints on storage, throughput, and resiliency. Lastly, we evaluate ARC's scalability, performance, resiliency, and ease of use.

The remainder of this thesis is organized as follows. Chapter 2 provides background information on the lossy compression algorithms and error-correcting codes we use. Chapter 3 presents related works and outlines the novelty of this thesis. Overall, the first three chapters present the motivation, background, and framing necessary for this work.

In Chapter 4, we detail our extensive soft-error fault injection study into lossy compressed data. We begin by outlining our experimental methodology. Following this, we analyze soft error effects on the decompression process, error-bounding capabilities, at various levels of loss, and on the resulting decompressed data's integrity.

Chapter 5 exhibits our solution to resolve the soft-error sensitivity present in lossy compressed data. In this chapter, we begin by laying out the design of ARC. We then move to analyze ARC's performance in terms of its scalability, performance, resiliency, and ease of use. Finally, in Chapter 6, we draw conclusions, discuss our findings further, and present future goals.

Chapter 2

Background

2.1 Lossy Compression

Researchers working with large datasets often use compression to reduce data size and resolve I/O bottleneck or memory issues [6, 29, 4, 16, 43]. While lossless compression algorithms, such as GZip [15] and ZStd [10], compress the data with no loss in precision, they are suboptimal for floating-point data due to entropy in the mantissa bits. As a result of this entropy, these algorithms can only achieve compression ratios just above 2× [34]. Error-bounded lossy compression algorithms, such as SZ [12, 39, 23] and ZFP [25], are a powerful alternative capable of achieving high compression ratios while maintaining an acceptable level of data quality. These algorithms achieve higher compression ratios by allowing a controlled amount of error into the dataset, which reduces the precision of the dataset and reduces issues with the high entropy mantissa bits. There are various approaches to bounding the introduced error among these algorithms, with each of these algorithms being studied extensively over various domains [28, 30, 8, 26, 5, 20, 40].

2.1.1 SZ

SZ is an adaptive block-wise prediction-based lossy compression framework currently in development at Argonne National Laboratory. Over the years, SZ has had three major iterations, 1.0 [12], 1.4 [39], and 2.0 [23]. The current version's (SZ 2.0) compression process begins by dividing the data into multiple non-overlapping blocks. SZ then begins a prediction process for each block

to determine which prediction method to use based on its features. SZ 2.0 offers three possible prediction methods with various strengths. First, it provides the traditional Lorenzo predictor from SZ 1.4. Next, it offers a mean-squared integrated Lorenzo predictor, which operates best on data values clustered to a small interval. Lastly, it contains a linear regression-based predictor that derives and uses regression coefficients to predict values. Once it has predicted the values, it then categorizes them into various quantization intervals before compressing these intervals using Huffman encoding.

SZ currently provides three error-bounding modes, which we use in this thesis. The absolute error-bounding mode (SZ-ABS) uses the set error bound to limit the error within an absolute error range for each data value. As seen in Equation 2.1, the decompressed data value, d', will never be less or more than the original data value, d, minus or plus the set error bound, ϵ. Using this approach, SZ ensures that all data values experience the same fixed level of error.

$$d - \epsilon \leq d' \leq d + \epsilon \tag{2.1}$$

The point-wise relative error-bounding mode (SZ-PWREL) limits each data point's error by considering the data point's value. As seen in Equation 2.2, the decompressed data value will never be less or more than the original data value plus or minus the original data value times the error bound. Using this approach, SZ assumes larger data values are capable of tolerating more error while smaller data values require higher precision.

$$d - d\epsilon \leq d' \leq d + d\epsilon \tag{2.2}$$

Lastly, the Peak-Signal-to-Noise Ratio (PSNR) error-bounding mode (SZ-PSNR) guarantees the decompressed data's PSNR rating retains a minimum set value. PSNR is a well-known metric used in the data reduction community to assess data distortion when using lossy compression. To calculate PSNR, one must first calculate the Root-Mean-Squared Error (RMSE) as seen in Equation 2.3, where N represents the data size while di and di' represent pairs of original and decompressed values, respectively. Using this result, one can calculate PSNR as seen in Equation 2.4. Using this approach, SZ prioritizes the overall decompressed data's integrity rather than any single data value.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(d_i - d_i')^2} \tag{2.3}$$

$$PSNR = 20 \times log_{10}(\frac{d_{max} - d_{min}}{RMSE}) \tag{2.4}$$

2.1.2 ZFP

ZFP is a block-wise transformation-based lossy compression framework currently in development at Lawrence Livermore National Laboratory [25]. ZFP's compression process begins by dividing the input data into fixed-size 4^d blocks, where d is the data dimensionality. It then aligns data values in each block to a common exponent by expressing each data value with respect to the largest floating-point exponent in the block, normalizing all data values in the block. Following this, ZFP converts the data to a fixed-point representation before using orthogonal block transformation to decorrelate the data of each block. This process produces transformation coefficients which ZFP then orders by expected magnitude before encoding the results one bit-plane at a time.

ZFP currently provides two error-bounding modes, which we use in this thesis. The fixed-accuracy error-bounding mode (ZFP-ACC) uses the user-set error bound to limit the error inside an absolute error range for each data value, similar to SZ-ABS. The fixed-rate error-bounding mode (ZFP-Rate) divides the data into 4^d sized blocks, where d is the data dimensionality. This mode then takes the product of a user-defined rate and 4^d to determine each block's compressed bit size. In this case, a lower rate leads to a higher compression ratio and less precision in the data, while a higher rate leads to the opposite. For example, given a 1-D array, each data block contains four values, and by setting the rate to 16, the four values are compressed and stored using only 16 bits. Among all the SZ and ZFP error-bounding modes we examine, ZFP-Rate mode is the only one to support random access into the compressed data. This capability is only possible as it decouples dependencies between 4^d sized blocks. However, as a trade-off, ZFP-Rate mode cannot achieve as high of a compression ratio as the other error-bounding modes and is similar to SZ-PSNR in that it does not bound the error of any given data value.

2.2 Error Correcting Codes

Researchers have created many different techniques to protect HPC applications and data from the critical issue of soft errors. These techniques include data redundancies, algorithm-based fault tolerance (ABFT), and error-correcting codes (ECC). Among these, reseearchers often use

ECC as it requires less space than redundancies and is application and data agnostic, unlike ABFT.

When using ECC, it is either applied at the hardware or software level. Hardware-level ECC, such as Chipkill, protects application memory and requires no extra work on the user's part. However, hardware-level ECC methods are not always available such as when working with OpenScience Grid computing systems. Furthermore, as we detail in Chapter 4, lossy compressed data is sensitive to soft errors, and as this data remains in storage for long durations, the level of protection provided by the storage system may be insufficient for the user. With this in mind, we implement all ECC algorithms at a software-level to ensure the protected lossy compressed data has adequate protection that is available in any environment. In this thesis, we utilize four different ECC algorithms.

First, single-bit ***Parity*** codes work by using an individual bit to guarantee an even number of bits are set to 1 in a given bitstream. If there is an even number of bits set to 1, the parity bit is set to zero. Conversely, if an odd number of bits are set to 1, the parity bit is set to one, making the number of bits set to 1 an even quantity. After calculating the parity, if any bit flips, it will produce an odd number of bits set to 1, indicating a fault is present.

Second, ***Hamming*** codes work by using a parity check matrix and a syndrome to detect and correct any single bit errors in a given bitstream. The size of the parity check matrix depends on the bitstream size, with the overhead decreasing as the bitstreams size increases. Once the algorithm has calculated the parity check matrix, if any bits flip, this will produce a syndrome that the algorithm uses to identify and fix the single-bit error.

Third, ***SEC-DED*** (Single-Error Correct Double-Error Detect) codes are a variation of Hamming codes that include an additional parity bit. This extra parity bit enables the algorithm to detect and correct single-bit errors and detect double-bit errors. Once the algorithm has calculated the parity check matrix and additional parity bit, if any single bit flips, the additional parity bit becomes incorrect, producing a syndrome that the algorithm uses to identify and fix the single-bit error. However, if any two bits flip, the additional parity bit will still appear correct, but this will still produce a syndrome, notifying the algorithm that a double bit error is present.

Fourth, ***Reed-Solomon*** code is the strongest ECC algorithm we use and work by breaking the data into blocks, called data devices. The algorithm uses these data devices to create parity code devices. Overall, a Reed-Solomon code is capable of correcting m corrupted devices, where m equals the number of previously produced parity code devices. While this makes Reed-Solomon capable of correcting many isolated and burst errors, it also introduces a high overhead.

Chapter 3

Related Work

3.1 Fault Injection Studies

With every new HPC system becoming even grander, soft errors are becoming even more commonplace on these systems. As a result, many studies have investigated the cause and impacts soft errors have on applications and data [38, 35, 13, 37, 33, 36, 19, 31]. While node location and temperature affect the possibility of encountering soft errors, these studies show that the root cause and failure rates depend on the system [35]. However, while understanding soft errors is critical to improving resiliency, few works have studied soft error effects on compressed data.

In 2017, Avramenko et al.'s study examined how soft errors affect a set of three lossless compression algorithms to compare their robustness [3]. The authors develop a run-time environment that compresses the data before the fault injection environment injects a soft error into it to test these algorithms. Following this, the fault injection environment classifies the results and evaluates the effect of the injected soft error. In 2020, Li et al. analyzed SZ's internal subroutines to determine their susceptibility to SDC [21]. Specifically, they evaluate SZ's regression coefficient calculation, best-fit predictor selection, data prediction, decompressed data calculation, Huffman encoding, and lossless compression steps. From their evaluation, the authors implemented various error tolerance methodologies. Also in 2020, Shan et al. developed a lossy compression fault injection tool called LCFI [32]. LCFI uses error distribution-based fault models to simulate soft errors in lossy compressed data to assess the effects the error has on the application results.

While all of these works share the goal of better understanding the effects soft errors have on

compressed data, they are all limited in some capacity. First, Avramenko et al.'s work only focuses on lossless compression algorithms. Second, Avramenko et al.'s and Li et al.'s studies only focus on understanding soft error effects on the compression process itself. Third, Shan et al.'s work only simulates soft errors and focuses on the impact of the soft error on the application using the lossy compressed data. This thesis is more expansive as we focus on the compressed data itself, aim to protect both lossless and lossy compressed data, and perform an extensive soft error fault injection study into lossy compressed data. To the best of our knowledge, this thesis is the first work to provide this viewpoint, and when joined with prior studies, one is capable of achieving a complete understanding of soft error sensitivities in compressed data.

3.2 Soft Error Resiliency Works

As researchers cannot prevent soft errors completely, application and data resiliency are vital. As a result, many prior studies focus on improving resiliency on HPC systems [7, 42, 14, 17, 11]. While each approach is different, these various studies all seek to produce a more error-resilient system. However, few works have worked towards creating more resilient compressed data.

In 2005, Nguyen et al. developed fault-tolerant error-detecting approaches for the major subsystems of the JPEG 2000 image compression standard [27]. In this work, the authors protect each subsystem using various methods, including binary decision variables and cyclic redundancy check parity values. From their results, the authors find their technique achieves high error-detecting capabilities while only incurring a slight overhead.

In 2020, Li et al. created an SDC resilient version of SZ that includes protection for SZ's internal operations [21]. In this case, the authors protect the internal operations by breaking the data into blocks to reduce the possibility of SDC propagation, use checksums to detect and correct errors in sensitive internal processes, and use redundant instruction at specific points to handle computation errors. Upon evaluating their approach, the authors found their changes introduce only a 10% overhead.

In the same year, Li et al. also developed a series of data-analytic-based fault tolerance (DBFT) methods tailored explicitly for lossy compression algorithms [22]. They centered their design around the Adaptive Impact-driven SDC Detector (AID). Using AID, the authors estimate the current timesteps value using the previous three timestep values, and if the difference is greater

than the user-defined impact factor, the program rolls back to a previous timestep. Data writing can either be synchronous or asynchronous, so the authors propose three solutions depending on the situation. Upon testing these solutions, the authors find promising detection abilities with an overhead of 7.9%.

While these works aim to protect the compression process and compressed data, they all have some shortcomings. First, Nguyen et al.'s work and Li et al.'s SDC resilient SZ work aim to protect the internals of the compression process and, as such, are bound to that algorithm and require changing the internals of the compression algorithm. Next, Li et al.'s DBFT work protects the application from corrupt compressed data instead of shielding the compressed data itself. It also requires multiple timesteps to protect the data. Similar to these previous works, ARC (Automated Resiliency for Compression) aims to protect compressed data. However, it seeks to accomplish this in a decoupled black-box method that is not bound to a single current or future compressor and does not require any algorithmic changes.

Chapter 4

Evaluation of Soft Errors

Soft errors present a true threat to all applications and data found on HPC systems. This threat is especially true for lossy compressed data as a single soft error in the compressed data makes the decompressed data unusable, as our preliminary proof shows in Figure 1.1. Furthermore, to combat memory and I/O bottleneck issues, HPC data remains compressed for long durations, which further complicates this sensitivity.

In order to protect lossy compressed data from the threat of soft errors, we need to understand the impact of soft errors fully. Specifically, we must understand how soft errors impact the decompression process, the resulting data's integrity, and a compressor's ability to bound the error it introduces. Moreover, it is critical to understand how these impacts change with various levels of loss. We aim to answer these questions by examining how soft errors impact lossy compressed data.

4.1 Experimental Setup

4.1.1 Compressors

In this study, we assess the soft error resiliency of two industry-standard lossy compression algorithms: SZ 2.1.8.1 [23] and ZFP 0.5.5 [25]. When evaluating SZ and ZFP, we examine three of SZ's error-bounding modes and two of ZFP's, as we detail in Chapter 2.1. When evaluating SZ-ABS, SZ-PWREL, and ZFP-ACC, we use an error bounding value of $\epsilon = 0.1$. When assessing SZ-PSNR, we use a PSNR value of 90. When examining ZFP-Rate, we use a rate of 8. We use these

error bounding values in our study to maintain uniformity with previous works [23]. Lastly, we also adjust the error bounding values to evaluate soft error effects at different levels of loss. In this case, we manipulate the error bounding values to test compression ratios 50×, 25×, 13×, and 7×.

To streamline all uses of SZ and ZFP, we leverage the compression abstraction library LibPressio [41]. In development under Dr. Robert Underwood, LibPressio abstracts all user interactions with many leading lossless and lossy compression algorithms. LibPressio also normalizes the output of all compression algorithms and provides beneficial compression metrics.

4.1.2 Datasets and System

In the trials of our study, we utilize three production-level HPC datasets provided by SDRBench[1] that researchers commonly use in compression studies[12, 23]. The first dataset we use is the CESM global climate model sponsored by the National Science Foundation (NSF) and the U.S Department of Energy. From the CESM dataset, we use its 25.82 MB 2D CLDLOW variable data. The second dataset we use is the Hurricane Isabel dataset that visualizes the 2003 hurricane produced by the Weather Research and Forecast model, courtesy of NCAR and the NSF. From the Hurricane Isabel dataset, we use its 100 MB 3D pressure variable data. The third dataset we use is the Nyx dark matter simulation dataset. From the Nyx dataset, we use its 536 MB 3D temperature variable data. We chose to use these datasets in our study as they vary in size and derive from various scientific domains.

We use phase 9 nodes on Clemson University's Palmetto Cluster with Intel Xeon E5-2665 16 core CPUs and 128GB of memory when running our study trials.

4.1.3 Evaluation Process and Metrics

Each trial of our study begins by compressing one of the three corresponding datasets. Following this, we flip a single bit of the compressed data stored in application memory. When determining which bits to target, we use a uniform sampling approach to make the study tractable as exhaustively testing even a dataset resulting in between 1 million and 2.7 trillion trials. With this in mind, we target a 1%, 0.1%, and 0.01% uniform sampling of bits in the CESM, Hurricane Isabel, and Nyx compressed datasets, respectively.

[1] https://sdrbench.github.io/

After we flip a target bit, we attempt to decompress the data and, in the process, record the decompression return status. By recording the decompression return status, we are able to ascertain the percentage of trials that proceed with corrupt data, leading to silent data corruption (SDC) and error propagation. When the data decompresses successfully, we next examine the decompressed data to evaluate its integrity and accuracy in relation to the original dataset.

When evaluating the decompressed data's integrity, we utilize four metrics. The first metric we record is the *percent of incorrect elements*, which we define as the number of decompressed data values whose error violates the user-defined error bound. We collect this metric to ascertain the extent of error propagation in the decompressed data. It is worth noting that we do not collect this metric for trials using SZ-PSNR as this mode does not bound the error on a value basis. The second metric we record is the *maximum absolute difference* between the original and decompressed dataset. If corruption is not present, this difference stays within the user-defined error bound. We record this metric to gauge the magnitude by which the data corruption violates the error bound. The third metric we record is the *PSNR* rating of the decompressed data. When calculating PSNR, we use Equations 2.3 and 2.4, as we detail in Chapter 2.1. We collect this metric to assess the impact of the soft error on the overall data's integrity. The fourth metric we collect is the *decompression bandwidth* when decompressing the corrupted compressed data. By recording this metric, we uncover any changes in throughput caused by the soft error. Using this set of metrics, we are capable of identifying all changes in the decompression process and the resulting decompressed data quality.

4.2 Experimental Evaluation

4.2.1 Error Effects on Decompression Process

Upon completing all trials, we begin by examining each return status and group them into four categories: *Completed*, *Compressor Exception*, *Terminated*, and *Timeout*. When the compressed data successfully decompresses, even though an error is present, we mark the trial as *Completed*. This type of return status is the most dangerous as this ending result leads to a high risk of error propagation and SDC. When the compressed data does not decompress due to the compressor throwing an exception, we mark these trials as *Compressor Exception*. In these cases, the soft error interacts with the metadata found in the compressed data stream that the compressor uses during the decompression process. As such, the compressor notices something is amiss and throws an exception.

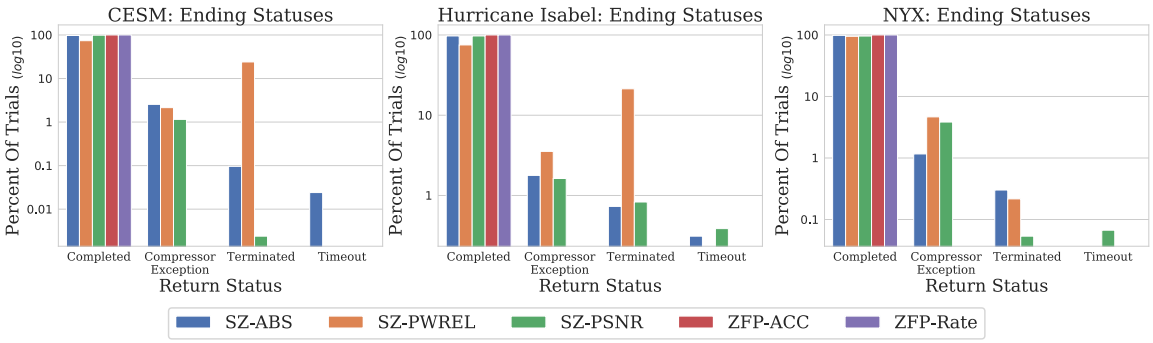


Figure 4.1: Distribution of return statuses for all fault injection trials.

In other cases, when attempting to decompress the compressed data leads to a segmentation fault or application crash, we mark these trials as *Terminated*. In *Terminated* trials, the soft error interacts with the compressed data unexpectedly, resulting in a situation the compressor cannot reconcile. While both *Compressor Exception* and *Terminated* trials do not lead to error propagation or SDC, they do lead to lost productivity and require the application to restart from a previous checkpoint or the beginning. Lastly, when a trial spends 3× the average time trying to decompress the data, we mark the trial as *Timeout*. When *Timeout* trials occur, they are due to corruptions in the decompression loop controlling metadata.

Figure 4.1 visualizes the distribution of return statuses that we found for all trials. From these results, we find 95.28% of all our trials *Completed* with only 4.72% of trials falling into one of the other three categories. Out of the other three categories, we found *Compressor Exception* trials occur most frequently in many cases with *Timeout* trials occurring least frequently. The only exception to this is SZ-PWREL which *Terminated* with a higher frequency on the CESM and Hurricane Isabel datasets. However, this is not true for all datasets, and we attribute this difference to the sample space we tested on.

Overall, we find these results concerning as *Completed* trials do not acknowledge the soft error and could lead to error propagation and SDC. Even more troubling, we find 100% of trials using ZFP *Completed*, meaning ZFP will never naturally catch soft error data corruption, which we attribute to the orthogonal block transformations it uses when working on the data. From these results, it is clear that compressed data must be protected from soft errors as any soft error impacts in compressed data have a high possibility of leading to SDC.

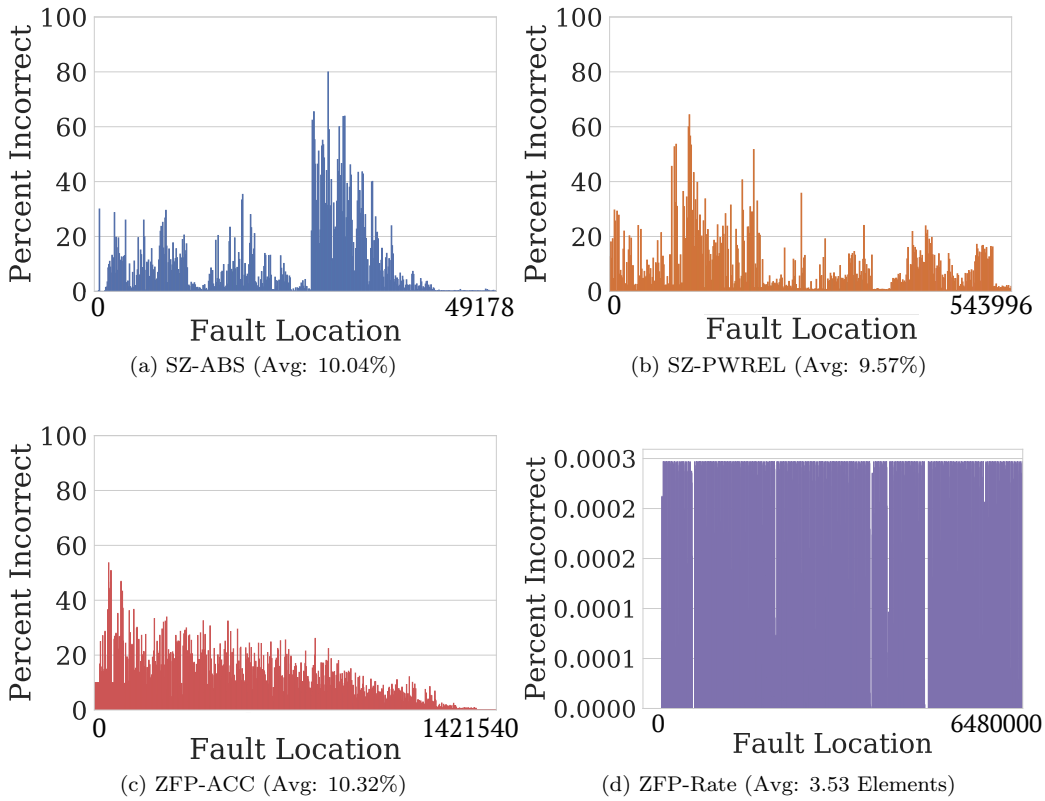
4.2.2 Error Effects on Error Bounding Capability

To continue our evaluation, we move to assess the impact a soft error has on a compressor's error bounding ability. In this evaluation, we only focus on *Completed* trials. For each of these trials, we calculate the difference between each corresponding original and decompressed data value to determine whether the decompressed value violates the user-defined error bound. Figure 4.2 visualizes the results for all CESM trials with the Hurricane Isabel and Nyx datasets displaying similar results.

Figure 4.2(a), (b), and (c) presents our findings for SZ-ABS, SZ-PWREL, and ZFP-ACC, accordingly. Upon evaluating SZ-ABS, SZ-PWREL, and ZFP-ACC, we find this percentage ranges from 0.01% – 80%, 0.03% – 64.4%, and 0.002% – 53.6%, respectively. While these results suggest that using ZFP-ACC leads to a lower percentage of incorrect elements and SZ-ABS or SZ-PWREL leads to a higher percentage, this is false. In actuality, these results show that a single soft error in any of the three leads to roughly an average of ∼10% incorrect elements with SZ-PWREL demonstrating a slightly lower percentage and ZFP-ACC demonstrating a slightly higher percentage. In Figure 4.2(c), there is a clear downward trend that occurs when using ZFP-ACC with soft errors in earlier bits leading to higher percentages. In contrast, in Figure 4.2(a) and (b), there is more variance with peaks throughout the bit locations when using SZ-ABS or SZ-PWREL. In either case, this is unsettling as, on average, a single soft error propagates to ∼10% of data values, leading to SDC that could further propagate or corrupt future calculations.

Unlike the other three modes, when evaluating ZFP-Rate mode, we find a single soft error leads to an average of only 3.53 incorrect data values, or 0.00005% of the data, that violate the error bound. Specifically, this value ranges from 0 – 16 incorrect data values, as seen in Figure 4.2(d). The low number of incorrect data values demonstrates the resiliency block compression algorithms have by removing dependencies between data blocks. In this case, ZFP-Rate mode divides the data into 4^d sized blocks or blocks containing 16 data values, preventing the error from propagating outside a given block. However, by dividing the data in this manner, ZFP-Rate mode only achieves a compression ratio of 4× while SZ-ABS, SZ-PWREL, and ZFP-ACC achieve compression ratios of 500×, 45×, and 17×.

In general, these findings indicate that when a soft error encounters lossy compressed data, it will propagate to ∼10% of the data, on average. These findings also indicate that certain compressed



(a) SZ-ABS (Avg: 10.04%)

(b) SZ-PWREL (Avg: 9.57%)

(c) ZFP-ACC (Avg: 10.32%)

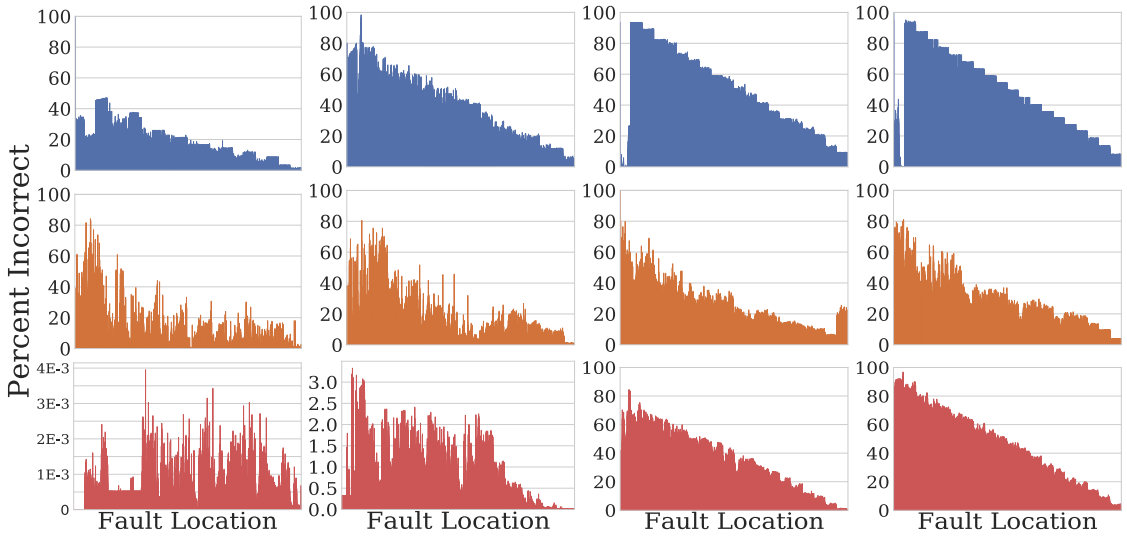
(d) ZFP-Rate (Avg: 3.53 Elements)

Figure 4.2: Percent of CESM values that violate the set error bound per fault injection location.

bits are more susceptible to soft errors. These bits are used to decompress multiple data values, and when the soft error impacts them, the error propagates to large portions of the data. Finally, while a fixed-rate compression algorithm like ZFP-Rate mode is more capable of containing soft errors, they are incapable of achieving the high compression ratios users often need from lossy compression.

4.2.3 Error Effects at Various Levels of Loss

To fully understand how soft errors affect lossy compressed data, we must also understand how soft errors affect data compressed to different levels. To accomplish this, we run further trials with each dataset and the SZ-ABS, SZ-PWREL, and ZFP-ACC modes. We exclude ZFP-Rate mode from these trials as we find uniform outcomes across all compression ratios. By adjusting the error bounding value we use in each mode, we obtain compression ratios of 50×, 25×, 13×, and 7× for each mode. Like before, we only visualize the CESM dataset results as we find similar results with the other datasets.



(a) CR: 50× (b) CR: 25× (c) CR: 13× (d) CR: 7×

Figure 4.3: Percent of CESM values that violate the set error bound per fault location at increasing levels of loss normalized by compression ratios (CR).
Top: SZ-ABS, Middle: SZ-PWREL, Bottom: ZFP-ACC.

Figure 4.3 displays our findings with the first row corresponding to SZ-ABS, the second to SZ-PWREL, and the third to ZFP-ACC. From these results, we find each row displays a similar trend, with higher compression ratios managing soft errors better, leading to lower percentages of incorrect elements. To obtain these higher compression ratios, we need to use less strict error bounds. These less strict error bounds are capable of tolerating more error in the data and mask small soft errors in the compressed data as compression errors. For instance, when using ZFP-ACC, we use error bounds of 10 and 0.5 to achieve compression ratios of 50× and 25×, respectively. With the average values in the CESM dataset being 0.3298, it is unrealistic that any researcher would use these error bounds as they introduce too much distortion into the data to be viable in a real-world situation. Therefore, while error propagation looks less likely when compressing the data more, in reality, the soft error is just masked. Furthermore, the error bound needed to achieve these compression ratios is often unsuitable for use with scientific data.

By studying both Figure 4.2 and 4.3, artifacts of the compression process also become visible. From these figures, it is clear that all SZ-ABS, SZ-PWREL, and ZFP-ACC graphs exhibit a similar downward slope when the compression ratio becomes low enough.

SZ's compression algorithm consists of multiple steps that make data predictions which it

then categorizes into various quantization intervals. As a final pass, SZ further compresses these intervals using a special lossless Huffman encoding algorithm. These figures display the resulting structure of this Huffman encoding process with elements needed to decode multiple other data values more often towards the start of the compressed data and more unique data values towards the end of the compressed data. While SZ does split the data into multiple non-overlapping blocks like ZFP-Rate mode, this final pass of Huffman encoding nullifies its ability to contain soft errors to a single block.

On the other hand, ZFP's compression algorithm first splits the data into equal-sized blocks, aligns them, and converts them to signed integers. It then uses near orthogonal block transformations to decorrelate block values before grouping the data by leading zeros, truncating it, and encoding each bit-plane one by one. When encoding the bit-planes, ZFP uses a custom group testing encoding procedure to encode their transformation coefficients. This type of encoding uses significance tests to continue to split the coefficients down further. We find the resulting structure of this process in the figures with more significant parts of the data towards the front of the compressed data and less significant chunks towards the end of the compressed data.

4.2.4 Error Effects on Data Integrity

To complete our soft error assessment, we evaluate how soft errors affect the decompressed data's integrity. To gauge the effects soft errors have, we document the decompression bandwidth, the maximum absolute difference, and PSNR for all trials marked as *Completed*. We visualize our findings in Figure 4.4 which displays the mean and standard deviations for all trials utilizing each configuration. In this figure, we also show a control case for each corresponding trial to facilitate comparisons.

4.2.4.1 Decompression Bandwidth

The first metric we evaluate is the effects of soft errors on decompression bandwidth. Our results show that the average decompression bandwidth of all soft error trials is relatively close to their corresponding control trials. However, while the soft error trials are close, in all cases, their decompression bandwidth is slightly lower than the control trials. Furthermore, we also see the standard deviation of all soft error trials is higher than the control trials. These differences indicate that while the soft errors do not stop the decompression process, they do make it more unstable.

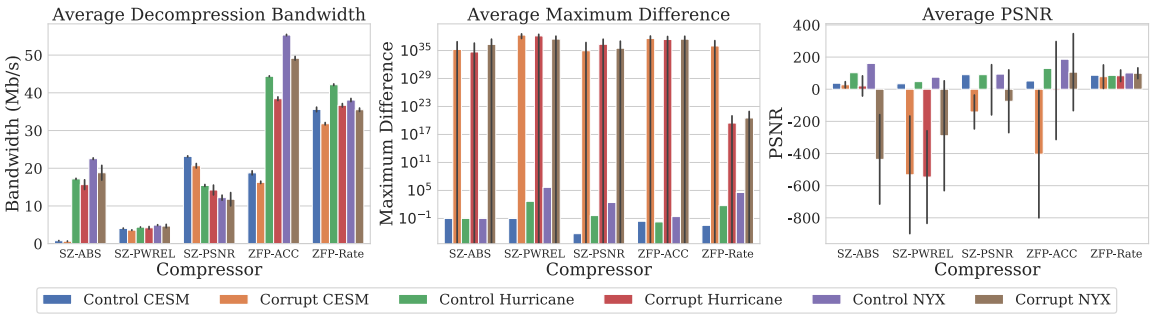


Figure 4.4: Average data integrity metrics for all fault injection trials.

This instability is due to changes in data values that the decompression process does not expect, which causes slight shifts in the decompression bandwidth.

In some rare trials, the decompression bandwidth we find is either extremely high or low. These rare instances are due to the soft error interacting with critical metadata in the compressed data stream that manages the decompression process. By interacting with these essential bits, the decompression process either ends earlier than expected or loops near infinitely. We observe that, while rare, instances of extraordinarily high or low decompression bandwidths are a possible indication that soft errors have impacted the data.

4.2.4.2 Maximum Absolute Difference & PSNR

The next metric we evaluate is the effects soft errors have on each trial's absolute maximum difference and PSNR. Looking at our absolute maximum difference results, we find the average maximum difference of all soft error trials vastly surpasses the user-defined error bound. Specifically, across all soft error trials, we find the absolute maximum difference is on the order of 10^{19} – 10^{38} instead of being less than 0.1 for trials using SZ-ABS. This orders-of-magnitude change is due to the soft error interacting with bits that the decompression process uses later to rebuild the more significant bits of data values. For example, the biggest changes occur when the soft error affects bits needed to reconstruct the exponent bits of data values. However, we find that these orders-of-magnitude shifts are not always guaranteed when looking at the standard deviation. Instead, changes to the maximum absolute difference vary wildly, and as such, one cannot expect orders-of-magnitude changes to happen every time a soft error occurs. Still, if an orders-of-magnitude shift occurs, this is a possible indication that soft errors have affected the data.

Upon reviewing the PSNR rating of all soft error trials, we find significant drops in a majority

of trials. These drops in decompressed data quality are due to error propagations in the data due to the single soft error. However, this does not happen when using ZFP-Rate mode due to its block-based compression approach. Since ZFP-Rate mode removes dependencies between blocks of data, the error cannot propagate to other blocks of data. As a result, ZFP-Rate mode retains a higher PSNR rating than the other error-bounding modes. Upon analyzing the standard deviation of all soft error trials, we determine that the PSNR rating is not easy to predict as it fluctuates depending on the error location, similar to our findings with the absolute maximum difference. This fluctuation is reasonable as the PSNR depends on the percent of incorrect elements and the maximum absolute difference. Nevertheless, an insufficient PSNR rating is an indication that soft errors have possibly interacted with the compressed data.

4.3 Error Effect Observations

The findings of our study provide us with a more in-depth knowledge of how soft errors interact with lossy compressed data, and from these findings, we draw essential observations. After testing the various error-bounding modes of SZ and ZFP, we find none are error-resilient enough to defend themselves against soft errors on their own. Our findings demonstrate that 95.28% of all soft error trials *Completed* and did not apprehend the soft error during the decompression process. By not reacting, these trials result in the soft error propagating and causing SDC.

Our findings also confirm that the compressed bit a soft error interacts with affects many aspects of the resulting decompressed data. First, we find that the impacted compressed bit significantly influences the percentage of incorrect elements in the decompressed data. When testing various levels of loss at different compression ratios, we found similar trends between all error-bounding modes we evaluate. Specifically, we find that more sensitive bits appear towards the beginning of the data more frequently when using a more strict error bound. Using a less strict error bound masks the presence of soft errors in most cases, but the required error bound is not always suitable for scientific data. Overall, these results confirm that certain bits are more sensitive to soft errors. While it would be advantageous to protect only these bits, the location of these bits depends on a range of factors, making it cumbersome to defend them only. As a result, it is more beneficial to provide equal protection to all bits.

When evaluating the resulting decompressed data's integrity, we find the soft error impacts

the decompression bandwidth, absolute maximum difference, and PSNR in various ways. First, our findings show that soft errors do not affect the decompression bandwidth in a significant way. Still, in all trials, the decompression bandwidth is slightly lower on average when soft errors are present. Next, when examining the decompressed data quality, we find that the absolute maximum difference and PSNR both drastically change for the worse on average when a soft error is present. While a very high decompression bandwidth or absolute maximum difference and a very low decompression bandwidth or PSNR are possible indicators of soft error impacts, these outcomes are not always likely to occur. As a result, one cannot count on these metrics to always signal soft error corruption in the compressed data.

On the other hand, our findings also confirm ZFP-Rate mode as the most error-resilient error-bounding mode. In all of our findings, ZFP-Rate mode always prevents the soft error from propagating outside of the block it impacts. This prevention is possible due to ZFP-Rate mode's fully decoupled block-based compression approach. Specifically, we find the number of incorrect elements never exceeds the block size while its PSNR rating always retains near original levels. However, as we detail in Chapter 2.1, ZFP-Rate mode is incapable of achieving high compression ratios like the other error-bounding modes, which limits its use in a broader sense.

In general, we find that neither SZ nor ZFP can manage soft errors in their current state. Although changing both to operate in a block-based compression approach will increase their resiliency, it will also reduce the compression ratio they are able to achieve. Yet, we must do something as a single soft error to either in their current state leads to an average of $\sim$10% of the decompressed data becoming incorrect.

Chapter 5

Improving Resiliency

Using our findings from the previous chapter, we develop an approach to shield lossy compressed data from soft errors. The results from our last chapter indicate that most soft errors go undetected during the decompression process. Therefore, we must ensure that no unresolved soft errors have impacted the lossy compressed data before decompressing it. Furthermore, as determining the most sensitive bits is cumbersome, we must provide equal protection to all compressed bits.

Current conventional techniques for data protection include creating data redundancies, algorithm-based fault tolerance (ABFT), and error-correcting codes (ECC). ECC is a popular choice as it requires less storage than keeping redundancies of the data and is application and data agnostic, not like ABFT. Nevertheless, there are many different ECC algorithms, and without prior knowledge, determining the correct algorithm is challenging.

In many cases, systems provide hardware-based ECC, such as Chipkill, to provide protection for the memory of applications running on them. However, hardware-based ECC is not always available in all circumstances, including in OpenScience Grid computing situations. Furthermore, lossy compressed data remains in storage for long periods of time, and the protection provided to storage may be insufficient for the user. Due to these concerns, we use a software-based approach when designing our protection scheme. Specifically, this approach ensures lossy compressed data has adequate protection that is constantly available.

To alleviate the burden of choosing the right ECC from the user and mitigate the shortcomings of hardware-based solutions, we develop ARC (Automated Resiliency for Compression) as a portable software-based solution to facilitate the protection of lossy compressed data.

5.1 ARC: Automated Resiliency for Compression

ARC's design consists of two major components. The front-end component and the primary point-of-contact users have with ARC is through the ARC Interface. The second essential component where ARC performs all encoding and decoding is the ARC Engine. However, before users begin using ARC, the data they wish to encode must be in the form of a `uint_8` byte array. ARC operates in this manner as most lossy compression algorithms return the compressed data in this form. Moreover, by generalizing the input to be a `uint_8` byte array, any data can use ARC to provide protection as long as the user can represent the data in this form. For example, this approach enables ARC to protect standard data arrays along with lossless and lossy compressed data.

5.1.1 ARC Interface

Before using the front-end ARC Interface functions, the user must first call ARC's initialization function, `arc_init()`, and provide it with the maximum number of OpenMP threads they want ARC to use. OpenMP [1] is a shared-memory multiprocessing programming interface consisting of an API and compiler directives that enable parallel programming through thread-level parallelism in shared-memory environments. OpenMP is essential to our design, as ARC uses OpenMP to apply thread-level parallelism to adjust the throughput of the various ECC algorithms it offers. This throughput adjustment is critical as different configurations of each ECC algorithm do not differ much in runtime. However, in many cases, researchers design applications to work with a certain amount of system resources. Therefore, providing ARC with a maximum number of OpenMP threads sets a resource utilization limit for ARC. Still, if there is no need for a limitation, the user can send `ARC_ANY_THREADS` to remove the thread limit restriction. Using the users input, ARC's initialization function loads necessary encoding information resources and starts the configuration training phase.

ARC's configuration training phase is a critical step that provides ARC with ECC configuration throughput estimations which ARC uses to provide accurate insight to the user on which ECC configuration is best to use. Before training starts, ARC checks the installation directory to see if cached configuration results are available and, if so, loads them. If this is the first time a user runs ARC on the system, ARC trains all possible ECC algorithms using an increasing number of

[1] https://www.openmp.org/

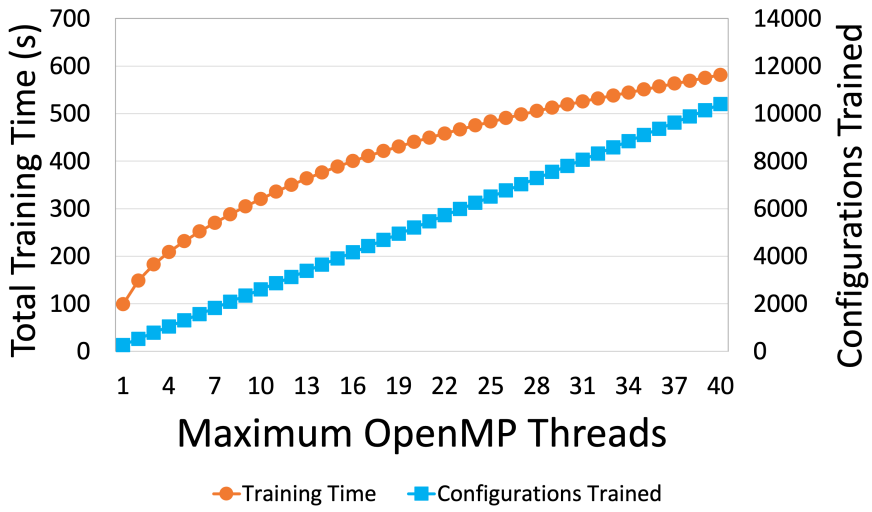


Figure 5.1: Training costs for various maximum OpenMP threads and resulting ECC configurations.

threads up to the maximum number of OpenMP threads. If ARC finds only a subset of configuration results, it loads what is available and trains using missing configurations. When training an ECC configuration, ARC uses the configuration with a simulated dataset and records the encoding throughput.

Figure 5.1 visualizes the required training time when a user initializes ARC using a various number of maximum OpenMP threads. We acquire these results using an Intel Xeon 6248G 20 core 2-Way Hyper-Threaded CPU with 372GB of memory. This figure shows that as we give ARC more threads, ARC generates more possible configurations. While training more configuration requires more time, they also improve ARC's ability to meet user constraints. This figure also demonstrates that the extra time ARC needs only increases logarithmically due to ARC using more OpenMP threads for each consecutive training step. Moreover, ARC only runs this process once for each number of threads and updates its training data through regular use. As a result, the training phase symbolizes a decreasing amount of ARC's total uptime with further use on a system. After users initialize ARC, it is ready to encode any `uint_8` byte array using its `arc_encode()` function, which optionally takes constraints on memory, throughput, and resiliency.

The first constraint the encode function takes is the memory constraint. This constraint serves an upper bound on the storage overhead ARC introduces while encoding the data. Specifically, ARC defines the storage constraint as the fraction of the byte array's size the user believes ARC should use when adding protection. However, if the user does not wish to impose an upper bound

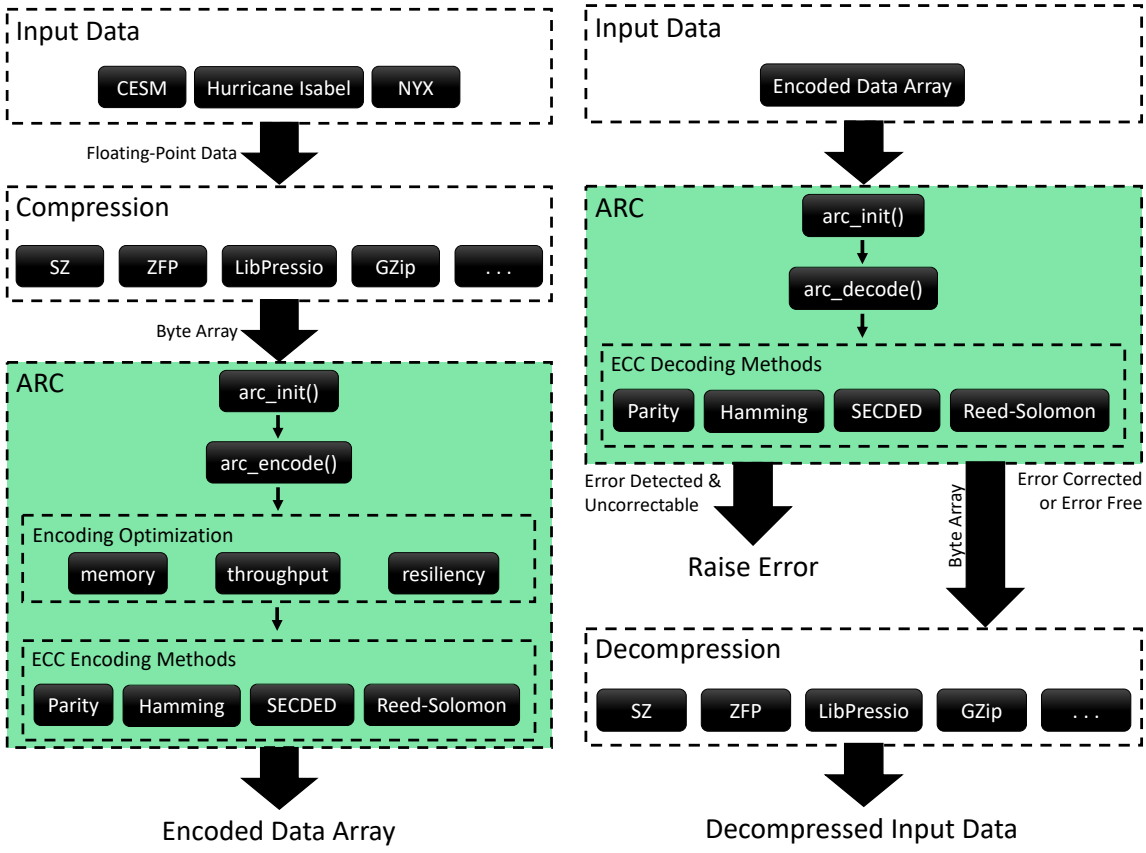
Resilience Constraint Input	Encoding Action
ARC_PARITY	Only use parity encoding
ARC_HAMMING	Only use Hamming encoding
ARC_SECDED	Only use SEC-DED encoding
ARC_RS	Only use Reed-Solomon encoding
ARC_DET_SPARSE	Only use ECC that can detect sparse errors.
ARC_COR_SPARSE	Only use ECC that can correct sparse errors.
ARC_COR_BURST	Only use ECC that can correct burst errors.
Expected Errors per MB	Only use ECC that can correct this number of uniformly distributed soft errors.

Table 5.1: Available ARC Resilience Constraint Options.

on the storage overhead, they can provide `ARC_ANY_SIZE` to remove this storage restriction. For example, if the user needs the input byte array's size not to increase more than 50% of its current size, providing a memory constraint of 0.5 will ensure ARC does not surpass this allowance. Overall, as ARC intends to work primarily on compressed data, the storage constraint ensures users are able to retain high compression ratios within their data.

The second constraint the encode function takes is the throughput constraint. This constraint serves as a lower bound on the time ARC takes while encoding the data. In particular, ARC defines the throughput constraint in units of MB/s. Though, if the user does not wish to impose a lower bound on the throughput overhead, they are able to provide `ARC_ANY_BW` which removes this restriction. For instance, if the user wants the encoding process to have a throughput of at least 100 MB/s, providing a throughput constraint of 100 ensures ARC uses the configuration with the necessary number of OpenMP threads to achieve this throughput. Since throughput varies from system to system, ARC uses the results from the training phase to help parameterize the model it uses for this constraint. In general, as ARC intends to have the smallest footprint on the host application, the throughput constraint enables the host application to maintain close to its original performance.

The final constraint the encode function accepts is the resilience constraint. This constraint acts as a filter by restricting which ECC algorithms ARC can use to only those the user chooses. This constraint takes an array of any combination of ECC method flag values, error-response flag values, or the number of expected uniformly distributed soft errors per MB of data. We display each of the available ARC resilience constraint options in Table 5.1. Using the ECC method flags, such as `ARC_SECDED`, restricts ARC to only the user-specified ECC method flags, making them ideal for users who already know which ECC methods they prefer. Utilizing the error-response flags, such as `ARC_COR_SPARSE`, guarantees ARC only uses ECC configurations capable of detecting or



(a) Encoding.

(b) Decoding

Figure 5.2: Overview of ARC.

correcting sparse/burst soft errors, which is ideal for users who have no background knowledge of ECC algorithms. Lastly, entering the number of expected uniformly distributed soft errors per MB of data ensures ARC uses only ECC configurations capable of correcting this number of errors. For instance, if the user predicts that a sixteenth of each MB of data will encounter a soft error, ARC will use Reed-Solomon encoding due to the higher possibility that burst errors will occur. Conversely, with lower soft error rates, ARC will use Hamming or SEC-DED as burst errors will not be as likely. This input is the most abstracted approach as users can discuss with their systems admin to determine this value. Primarily, as ARC must provide enough protection for each user, the resilience constraint guarantees the desired resiliency level to the data.

ARC uses each of these three constraints to encode the data as we visualize in Figure 5.2a. ARC's encoding process starts by passing the three constraints to its encoding optimization func-

tions. Within these functions, ARC first uses the user-specified resiliency constraint array to filter the potential ECC configurations it should choose from down to the user-defined set. Following this, ARC scans through the resulting set of configurations to create two subsets of configurations. The first subset is of all configurations whose memory overhead is less than but closest to the user-specified memory constraint. The second subset is of all configurations whose throughput is above but closest to the user-specified throughput constraint. ARC then compares each of these subsets to find a configuration that best meets the requirements of both. Suppose ARC doesn't find a configuration that meets these requirements. In that case, it uses the ECC configuration with the memory overhead closest to the memory constraint while using a number of OpenMP threads to achieve as close to the throughput constraint as possible. ARC passes this configuration information to the ARC engine, which encodes the byte array before passing it back to the user via the ARC Interface.

To access the encoded data, user must first call ARC's decode function, `arc_decode()`, as we demonstrate in Figure 5.2b. When a user calls the decode function, the ARC interface passes the encoded data to the right ARC Engine decoding function. While decoding each block of data, this function first checks each for any soft errors. If the decoding function detects and cannot correct a soft error, ARC reports an error to the user. Conversely, if ARC finds no errors or is capable of correcting them all, it repairs all errors, decodes the data, and returns the decoded data to the user.

Before the application completes or when the user no longer needs ARC, they must call the `arc_close()` function. This function operates as both an update step and a clean-up step. First, this function calls the `arc_save()` function, which updates all cached configuration information with up-to-date iterations that ARC gathered during normal operations. Then ARC removes all necessary memory allocations. This step is critical as this ensures any fluctuations in ARC's throughput are recorded, ensuring all estimations remain accurate for further uses of ARC.

5.1.2 ARC Engine

While we do provide the ARC interface as a front-end, users are also able to interact with ARC more directly through the ARC Engine functions we detail in Table 5.2. The functions that comprise the ARC Engine consist of two major categories. The first set of functions is the constraint optimization functions, while the second set is the encode/decode functions.

The constraint optimization functions include the functions ARC uses to determine which ECC configuration is optimal given the user's constraints. First, the `arc_memory_optimizer()`

Functions	
arc_memory_optimizer()	arc_hamming_decode()
arc_throughput_optimizer()	arc_secded_encode()
arc_joint_optimizer()	arc_secded_decode()
arc_parity_encode()	arc_reed_solomon_encode()
arc_parity_decode()	arc_reed_solomon_decode()
arc_hamming_encode()	

Table 5.2: Available ARC Engine functions.

takes a memory constraint and a resiliency constraint which it compares to existing configuration information to determine which ECC configuration to recommend to the user. Next, the `arc_throughput_optimizer()` operates in the same manner but takes a throughput constraint and resiliency constraint instead. Finally, the `arc_joint_optimizer()` takes all three constraints and finds the best ECC configuration that satisfies all three constraints, like we discussed earlier. While ARC always listens to these recommendations when a user utilizes the ARC interface, users running these functions directly have the option to ignore these suggestions for any reason

The encode/decode functions comprise four pairs of encode/decode functions for each ECC algorithm we detail in Chapter 2.2. The first set of functions are for encoding and decoding with single-bit even parity. These functions apply a minimal level of resiliency to each equally-sized block of data through a single parity bit, which reduces the possibility of SDC. When using these functions, the user needs to provide the number of data bytes each parity bit should protect. Even though this approach introduces the least overhead and detects all odd multi-bit errors, it is helpless against even multi-bit errors.

The second and third sets of functions are for encoding and decoding with standard Hamming or SEC-DED. Each set of functions generate several parity bits for either one byte or eight bytes of data at a time. While each set of functions has slightly more overhead than parity, they are both capable of detecting and correcting single-bit errors, with SEC-DED capable of also detecting two-bit errors. Although each set of functions are capable of correcting errors, they are both helpless against burst errors.

The final set of functions are for encoding and decoding with Reed-Solomon encoding. These functions leverage the Jerasure erasure coding library [2], which efficiently encodes while providing various erasure coding algorithms. These functions encode the data by dividing it into several eight-

[2]http://web.eecs.utk.edu/~jplank/plank/www/software.html

byte data devices before using them to construct a number of eight-byte code devices. By using these code devices, Reed-Solomon encoding is capable of detecting and correcting any number of corrupt devices up to the number of code devices created. Even though these functions provide the best protection of all ECC ARC provides and protect against burst errors, they require the largest storage overhead and have the lowest throughput. Overall, by providing this set of ECC algorithms, ARC offers a full range of protection levels to cover any user needs.

By exposing the ARC Engine, we also provide users with more control over how they protect their data and facilitate the seamless integration of ARC directly into other applications. For instance, by using ARC as a final step in any lossy compression algorithm or a library such as LibPressio, the resulting lossy compressed data is exposed for only a short period of time. Furthermore, the ECC algorithms we present in this paper do not represent a limitation for ARC. Instead, ARC supports additional custom ECC algorithms and user constraints to sustain user needs.

5.2 Evaluating ARC

When designing ARC, we aimed to satisfy four main goals with its final design. First, as ARC aims to solve soft error problems on HPC systems, it must operate efficiently and scale efficiently when provided with more resources. Second, as each user's needs are unique, ARC must satisfy them all adequately. Specifically, ARC must determine the best ECC configuration to satisfy each constraint when given constraints on resiliency and storage or throughput overhead. Third, as ARC's primary goal is to prevent soft errors in lossy compressed data, it must provide enough protection so that these errors do not go unresolved. Finally, as ARC aims to simplify the protection of lossy compressed data, it must be easy to use, require minimal code changes, and introduce the smallest overhead possible. Therefore, when evaluating ARC, we examine its scalability, performance, resiliency, and ease of use.

5.2.1 Scalability Evaluation

Since ARC aims to solve soft error issues on HPC systems, it is critical that it scales efficiently so that the host application maintains its efficiency. To understand how ARC scales with further resources, we must evaluate how each ECC algorithm scales with extra OpenMP threads. When assessing each ECC algorithm's scalability, we run our trials on an Intel Xeon 6248G 20 core

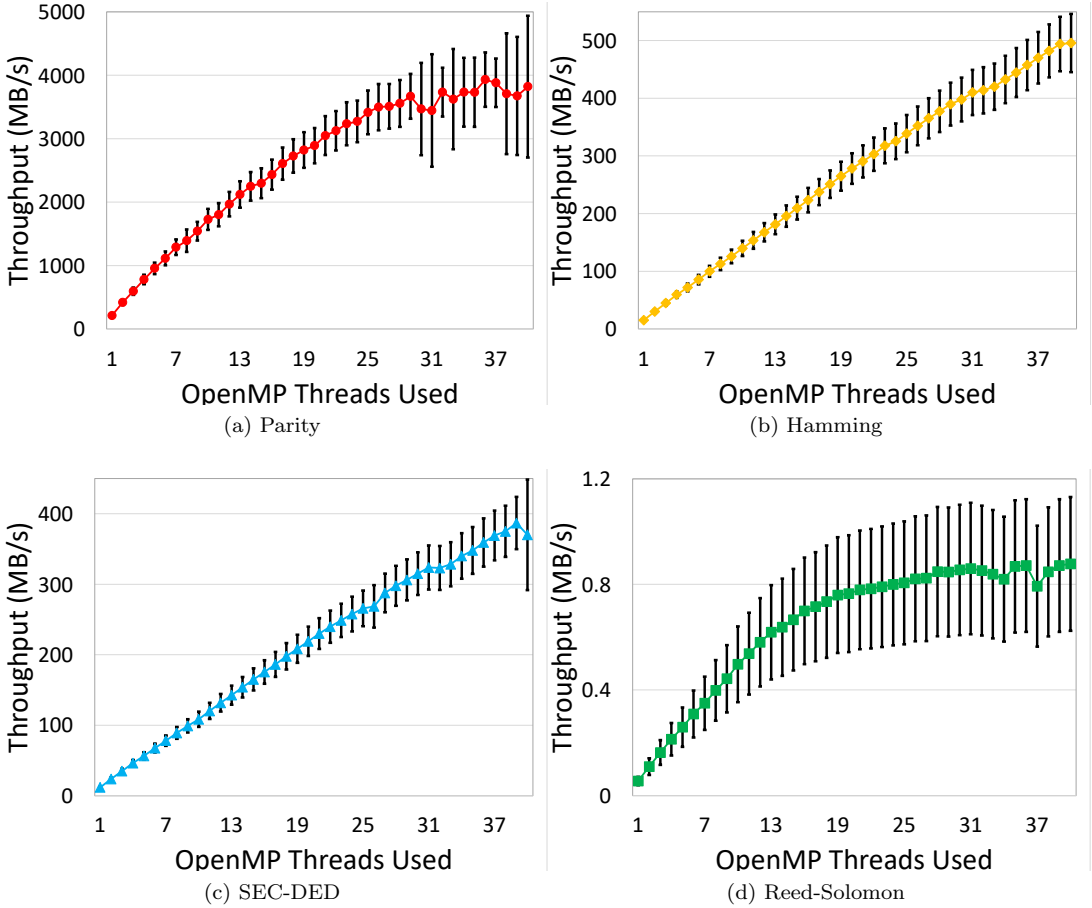


Figure 5.3: ARC's encoding scalability.

2-Way Hyper-Threaded CPU with 372GB of memory and set the maximum number of OpenMP threads for ARC to 40. For each ECC algorithm, we use the CESM dataset and run ten trials with each number of threads.

Figure 5.3 and Figure 5.4 display our findings for all encoding and decoding processes. Upon analyzing Figure 5.3, we find nearly all encoding processes scale near linearly when provided with extra threads. Specifically, we find the throughput of all methods ranges from 0.04 – 3730 MB/s. When comparing the 40 to 1 thread performance improvement, we find Parity, Hamming, SEC-DED, and Reed-Solomon exhibit speedups of 19.7×, 26.8×, 33.9×, and 16.4×, respectively. Upon analyzing Figure 5.4, we find similar results for the decoding processes. With these processes, we find the throughput of all ranges from 10.64 – 3602 MB/s. When comparing the 40 to 1 thread performance improvement, in this case, we find Parity, Hamming, SEC-DED, and Reed-Solomon

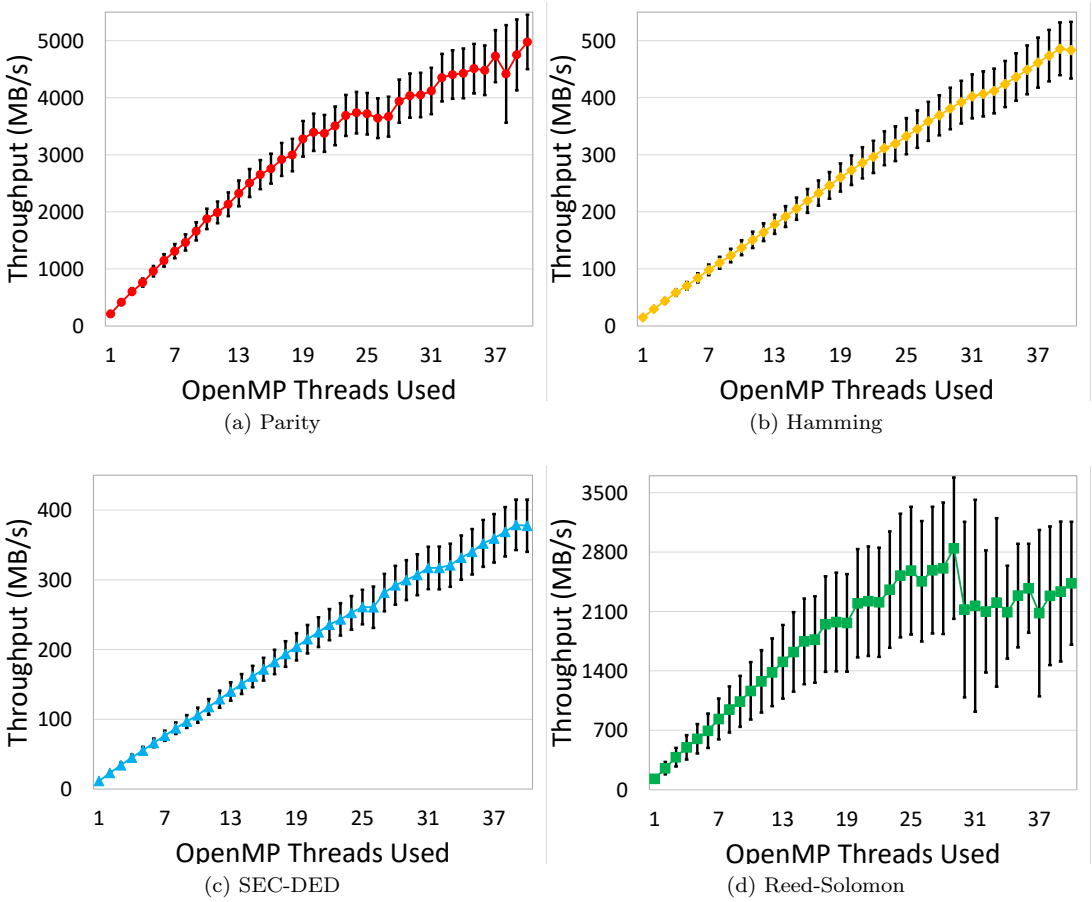
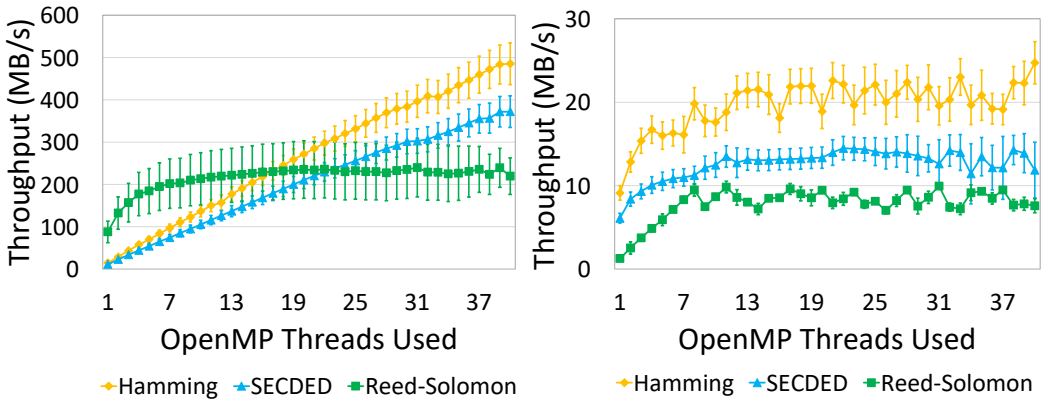


Figure 5.4: ARC's decoding scalability.

display speedups of 18.6×, 33.5×, 33.5×, and 18.3×, respectively.

From our findings, we observe that using a wide range of OpenMP threads leads to a wide range of throughputs being available to ARC as each of the different ECC algorithms scales differently in parallel. While we find the throughput variance increases as we add more OpenMP threads, this is not an issue for ARC. After each use, ARC updates its configuration information to ensure it is always capable of accurately choosing the optimal ECC configuration for the user. Upon comparing ARC's range of throughputs to that of SZ and ZFP, we determine that ARC is more than capable of keeping pace with the less than 200 MB/s throughputs each display [24]. Still, repairing soft errors requires extra computations during the decoding process, and, as such, we must ensure the decoding process continues to scale while it fixes soft errors.

In order to evaluate how soft errors impact the decoding process, we conduct two separate



(a) 1 Correctable Error

(b) 100,000 Correctable Errors

Figure 5.5: Effect of 1 and 100,000 correctable soft errors on ARC's decoding throughput.

experiments. In the first experiment, we inject a single correctable soft error into the encoded data blocks to simulate a simple single-bit soft error. On the other hand, in the second experiment, we inject 100,000 random correctable soft errors into the encoded data blocks to simulate an unlikely worst-case scenario. When injecting the soft errors in both experiments, we randomly inject the error into the encoded data blocks while ensuring that it is correctable to evaluate the correction processes throughput. We do not evaluate single-bit parity within these experiments since it can only detect soft errors and not correct them.

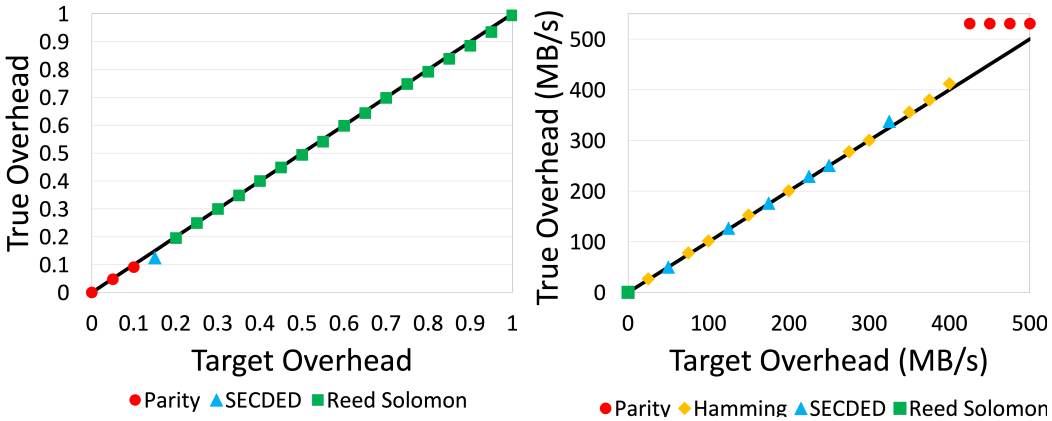
Figure 5.5 illustrates our findings when a single soft error and 100,000 soft errors are present. Upon analyzing these figures, we find that a single correctable soft error only disrupts the Reed-Solomon decoding throughput. This disruption is due to the high repair cost associated with Reed-Solomon encoding and causes the speedup with 40 threads to drop from 18.3× to 2.7×. By comparing the single correctable soft error trials to the 100,000 correctable soft error worst-case trials, we find drastic drops in the throughput of all three ECC algorithms. Specifically, we find the 40 thread performance improvements of Hamming, SEC-DED, and Reed-Solomon drop to 2.64×, 2.43×, and 1.1×, respectively. Still, even after these large degradations in throughput, each maintains a manageable throughput above 7 MB/s while correcting the data. Consequently, it would take more correctable soft errors than one would ever likely see to reduce the decoding throughput to such drastically low levels. Instead, in the more likely case that a single correctable soft error occurs, ARC's decoding processes maintain close to their original throughput, with Reed-Solomon being the only exception by dropping 6.7× due to its higher repair costs.

5.2.2 Performance Evaluation

With each user's needs being unique, it is critical that ARC is able to satisfy user needs efficiently. To assess how well ARC is able to meet user constraints, we must evaluate how ARC works to meet user constraints on storage and throughput when it has access to any ECC algorithm and when it is limited. When evaluating ARC, we use the CESM dataset using SZ-ABS and an error bound of $\epsilon = 0.1$ with other datasets yielding similar results. We also set the maximum number of OpenMP threads to 40 and use the same system as we do in the previous section.

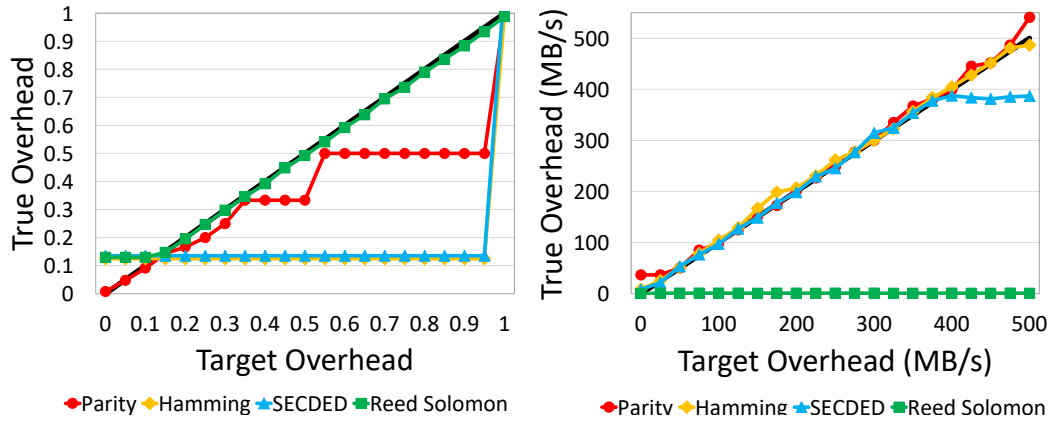
Figure 5.6(a) illustrates ARC's performance when ARC has access to any ECC algorithm while attempting to satisfy a user-specified memory constraint. This figure shows that ARC manages to use the best ECC configuration that most optimally uses the provided space given by the upper bound on storage. For instance, when the user wants the input data's size not to increase more than 20% and provides a memory constraint of 0.2, ARC chooses Reed-Solomon encoding with 15 code devices protecting every 241 data devices, resulting in a memory overhead of 19.5%. However, when the user provides ARC with a higher memory constraint of 0.9, ARC leverages the extra storage to improve its protection by choosing Reed-Solomon encoding with 103 code devices protecting every 153 data devices, leading to a memory overhead of 88.5%. In all cases, ARC switches between all available ECC algorithms to utilize the available storage budget best.

However, when a user limits ARC to any single ECC algorithm, it cannot always utilize the entire storage budget. Figure 5.7(a) demonstrates this effect by showing ARC's performance when the resiliency constraint limits ARC to any single ECC algorithm. Upon analyzing this figure, we find ARC always uses the chosen ECC algorithm with the configuration that best uses the provided space given by the upper bound on storage. However, while limiting ARC to a subset of ECC algorithms guarantees a protection level, it also decreases ARC's ECC choices, and as a result, it cannot always use the entire storage budget. For example, ARC only provides two configurations of Hamming and SEC-DED, as we detail in Chapter 5.1.2, and as a result, cannot always use the entire storage budget. Similarly, Parity also demonstrates a step-like function since ARC only applies single-bit parity on a byte level. In other cases, ARC may need to go over the memory constraint if it is too low and it only has access to a single ECC algorithm. For example, using a memory constraint of 0.05 and a resiliency constraint that requires Reed-Solomon will force ARC to go over budget, display a warning, and use the Reed-Solomon configuration that results in the



(a) Memory Constraint (b) Throughput Constraint

Figure 5.6: `ARC_ANY_ECC` Performance Evaluation: Target Overhead vs. True Overhead.



(a) Memory Constraint (b) Throughput Constraint

Figure 5.7: Single ECC Performance Evaluation: Target Overhead vs. True Overhead.

lowest memory overhead possible. Overall, ARC will always use as much of the storage budget as possible while also avoiding going over the budget.

On the other hand, Figure 5.6(b) presents ARC's performance when ARC has access to any ECC algorithm while attempting to satisfy a user-specified throughput constraint. From this figure, we see that ARC always uses the ECC algorithm and configuration with a suitable number of OpenMP threads to maintain a desired throughput. For example, when given a throughput constraint of 0.5 MB/s, ARC uses Reed-Solomon over 15 OpenMP threads, resulting in a throughput of 0.51 MB/s. Yet, when given a throughput constraint of 300 MB/s, ARC is unable to use the slower Reed-Solomon encoding and instead opts to use SEC-DED over 34 OpenMP threads, resulting in

a throughput of 302.4 MB/s. While ARC could further improve the throughput by running with more threads in both cases, ARC chooses to use fewer threads to reduce its impact on the available resources, which is beneficial when system resources are highly contested in a host application.

However, like before, when a user limits ARC to any single ECC algorithm, it will not always be able to achieve the desired throughput. Figure 5.7(b) demonstrates this effect by showing ARC's performance when the resiliency constraint limits ARC to any single ECC algorithm. In this case, ARC applies the desired ECC algorithm with a configuration that best meets the lower bound on throughput. Again, while using the resiliency constraint does ensure a protection level, it also reduces ARC's options, and as such, it cannot always meet the necessary throughput requirements. For example, ARC's Reed-Solomon approach is not fast enough to meet most high throughput requirements. When ARC encounters a situation where meeting the throughput constraint is impossible for the desired ECC algorithm, it instead uses the configuration that gets as close as possible.

While we only show using the resiliency constraint with either the memory or throughput constraint in these figures, ARC also supports using all three in conjunction with one another. When using all three constraints, ARC first uses the resiliency constraint to filter its potential ECC configuration choices. Following this, the memory and throughput constraints will either work well together or conflict with one another. When they work well together and ARC is free to use any ECC configuration, we find results similar to those we display in Figure 5.6(a) and (b). For instance, when provided a memory constraint of 0.2 and a throughput constraint of 0.6 MB/s, ARC uses Reed-Solomon as both constraints make this possible. However, when the constraints conflict with one another, ARC works to balance a trade-off between the two that will satisfy both. For example, if provided a higher memory constraint of 1 and a higher throughput requirement of 100 MB/s, ARC uses SEC-DED instead of Reed-Solomon since Reed-Solomon is incapable of achieving the required throughput. Restricting ARC's potential ECC configurations further complicates this trade-off balancing process. When this happens, even when the memory and throughput constraints agree on a target ECC configuration, that particular ECC may not be allowed. As a result, ARC uses the ECC configuration from the set of potential ones that uses the available resources best. Overall, ARC still satisfies conflicting constraints to the best of its ability when using all constraints together but does not apply the highest level of protection.

5.2.3 Resiliency Evaluation

As ARC's main goal is to prevent soft errors from going unresolved, it must provide enough protection. To assess ARC's ability to provide protection, we must evaluate how ARC reacts to the presence of soft errors. When performing this evaluation, we use ARC to protect each of the previous datasets while providing a resiliency constraint of 1 error per MB and a free memory and throughput constraint. Using this input, ARC applies SEC-DED to every eight bytes of data to guarantee single error correction and double error detection. Following this, we rerun our fault injection trials from Chapter 4 using these new datasets. Our results from these trials indicate that ARC corrects all of the soft errors we inject. We anticipated this outcome as ARC implemented SEC-DED, but all ECC algorithms ARC provides are also capable of preventing single-bit soft errors from going undetected. However, these trials only use single-bit errors, which are not always the case when running on HPC systems.

While single-bit soft errors are the most common type of soft error on HPC systems, ARC must also be capable of protecting from multi-bit errors. To ensure ARC protects against these types of errors, users can directly choose SEC-DED or Reed-Solomon encoding through the resiliency constraint. Users can further optimize this protection by providing a higher storage budget and lower throughput requirements, both of which allow ARC to use stronger versions of the desired level of protection. For instance, by using `ARC_RS` and providing a memory constraint of 0.2, ARC uses a Reed-Solomon configuration with 15 code devices. However, by raising the memory constraint to 0.9, ARC uses a Reed-Solomon configuration with 103 code devices. While both approaches allow ARC to correct multi-bit and burst errors, the second approach allows ARC to correct more corrupt data due to the higher number of code devices. Conversely, if the user provides ARC with a memory constraint of 0.1, a throughput constraint of 700 MB/s, and does not specify a protection level, ARC will use single-bit even parity. With this ECC algorithm, ARC only detects single-bit and odd-numbered soft errors within each data block. Therefore, while ARC can provide adequate protection, it is up to the user to ensure ARC has the proper budget to protect the data.

5.2.4 Ease of Use Evaluation

As ARC aims to simplify the protection of lossy compressed data, it must be easy to implement. With this in mind, we demonstrate ARC's ease of use by displaying the necessary code

Algorithm 1 Integrating ARC

```
Input data Input uint8_t array
Input data_size Size of uint8_t array

arc_init(ARC_ANY_THREADS);

int err = arc_encode(data, data_size, ARC_ANY_MEM, ARC_ANY_BW, [ARC_ANY_ECC], 1,
encoded, encoded_size);
...
err = arc_decode(encoded, encoded_size, decoded, decoded_size);

arc_close();
```

changes to integrate ARC and describe how ARC could be used on the two recently decommissioned production-level HPC systems discussed in the study by Sridharan et al. [37, 35].

In order to deploy ARC within a host application, users must make various changes to their codebase to integrate ARC. We develop ARC such that the amount of code changes is as minimal as possible. We show the necessary four lines of code in Algorithm 1. Specifically, these lines include initializing ARC, encoding the data, decoding the data, and safely closing ARC. However, ARC only needs to be initialized and closed once per application run, so each internal use of ARC only requires two extra lines of code per encoding/decoding pair. This approach makes using ARC as lightweight as possible. Now that the user has integrated ARC into the host application, they must next determine which constraints to provide to ARC. While in this algorithm `ARC_ANY_MEM`, `ARC_ANY_BW`, and `ARC_ANY_ECC` represent the three constraints and provide the most robust ECC configuration, users may want to change these to satisfy their specific needs. The most appropriate way to choose these constraints is to consider the failure rate of the system the application is running on.

To illustrate how the failure rate should affect a user's constraint choices, we use the two systems discussed in the study by Sridharan et al. [37, 35] as an example. The first system in this study is Cielo which is an 8,500 node HPC system in Los Alamos, New Mexico, located at around 7,300 feet in elevation. The second system in this study is Hopper which is a 6,000 node HPC system in Oakland, California, located at 43 feet in elevation.

In these studies, the authors examine the failure rate per DRAM device over 30 days for each HPC system. Their findings show Cielo has nearly twice the failure rate as Hopper, which the authors attribute primarily to the difference in altitude between the two systems. Using the failure

rates of each system and the number of compute nodes found on each system, we calculate the mean time between failure (MTBF) for each HPC system. From our calculation, we find Cielo has a soft error failure every 1.9 days while Hopper has one every 5.43 days. Still, it is important to note that the failure rate we calculate does not include undetected soft errors, which cause SDC. Thus, the true time between failure is likely lower.

In these studies, the authors also provide a full breakdown of every fault they find. On the Cielo system, they found that soft errors caused 34.9% of all faults, and on Hopper, they found that they caused 42.1% of all faults. Leveraging this knowledge, we determine that single-bit soft errors caused 70.79% of Cielo's faults, while we determine that single-bit soft errors caused 94.6% of Hopper's faults. With the failure rate of both systems and the distribution of soft errors that caused these faults, users are prepared to provide the most appropriate constraints to ARC.

For users running applications on Cielo, their applications will require more robust ECC protection due to the system's high failure rate along with the lower probability that a single-bit soft error was the root cause. Specifically, 29.21% of all faults on Cielo were not caused by single-bit soft errors and instead were caused by multi-bit soft errors. Upon breaking down these multi-bit soft errors further, we find most of them occur as burst errors in the same DRAM device. Using this information, applications on this system need the higher level of protection that the Reed-Solomon algorithm provides. To make sure that ARC uses this ECC algorithm, the user can use the `ARC_COR_BURST` flag, the `ARC_RS` flag, or provide a higher memory constraint and lower throughput constraints, as we discussed in Chapter 5.1. Alternatively, the user could also utilize the ARC Engine directly and use the Reed-Solomon encoding and decoding functions. By utilizing one of the flags and easing the constraints more, ARC provides further protection by using even stronger Reed-Solomon configurations, as discussed in Chapter 5.2.3.

Conversely, users running applications on Hopper do not need as strong of an ECC algorithm as users running on Cielo. This difference is due to Hopper having a much lower failure rate than Cielo, along with the fact that single-bit flips cause over 90% of soft errors. Furthermore, by breaking down the few multi-bit soft errors found on Hopper, we find that only 4.05% of these occurred as burst errors and are spatially close to one another. Using this information, it is clear that the more robust ECC algorithms are unnecessary in most cases when running on this system. Therefore, by talking with a system admin and entering the predicted number of errors per MB or using the ECC algorithm they recommend, ARC will provide the necessary protection to the data. Following this,

users can then use the memory and throughput constraints to tune ARC to their specific needs further. For example, if the user decides they want to detect and correct all single-bit errors, they can enter the predicted number of errors per MB. Alternatively, they can use the `ARC_COR_SPARSE`, `ARC_HAMMING`, or `ARC_SECDED` flags. Using either approach will ensure single-bit errors cause no interruptions.

Overall, our demonstration shows that integrating ARC into any application working with sensitive lossy compressed data only requires four lines of additional code. Our demonstration also illustrates that while ARC enables users to provide their ideal constraints, they should consider a system's failure rate and distribution of faults to set the most appropriate constraints for the system. With this knowledge, any user is capable of deploying ARC within their host application with ease.

Chapter 6

Conclusions and Discussion

In recent years, HPC systems have become immensely more powerful. By using these systems, researchers are able to solve previously intractable problems. The powerful applications they develop to solve these problems create massive datasets, causing stress on the I/O and storage subsystems. Moreover, while many areas of HPC systems have become more powerful, the I/O capabilities on these systems continue to lag behind and further exacerbate this bottleneck.

Researchers use data reduction methods to resolve issues caused by large quantities of data, including lossless and lossy compression. While lossless compression compresses data with no loss, it is not as effective as lossy compression due to the high entropy in floating-point mantissa bits. Therefore, researchers often use lossy compression to compress scientific data efficiently while controlling the amount of error introduced through an error-bounding mode and value.

As the presence of soft errors exists for all applications and data on HPC systems, users must vet any changes to an application to ensure the change will not lead to excessive soft error sensitivities. Figure 1.1 illustrates the soft error sensitivities found in lossy compressed data. Yet, even with this high sensitivity, very few works aim to assess and protect lossy compressed data from soft errors. This work provides an assessment and approach to protecting lossy compressed data from soft errors.

Upon completing our analysis of soft error sensitivities, we find various critical trends. After evaluating the soft error effects on the decompression process, we find 95.28% of all our trials *Completed*, leading to possible error propagation and silent data corruption (SDC). Moreover, we also found that 100% of our trials using ZFP *Completed*. From these results, we conclude that

current leading lossy compression algorithms rarely catch soft error data corruption. Following this, we also find that, on average, a single soft error propagates to ∼10% of data values. We find this trend exists among all leading lossy compression algorithms and fluctuates when compressing the data to different levels. Still, even at different compression ratios, we find sizeable portions of the data become incorrect when a single soft error goes unresolved in lossy compressed data. Finally, we find that a single soft error negatively affects the decompressed data's integrity. Specifically, we find the absolute maximum difference exhibits orders-of-magnitude shifts while the resulting PSNR significantly drops in most trials. The only exception to these results is the block-based ZFP-Rate compression mode that removes dependencies between data blocks. Specifically, we find the error never propagates outside the block it occurs in, resulting in less incorrect elements and better resulting PSNR. While wild changes in the decompression bandwidth, absolute maximum difference, and PSNR signify a soft error's presence, they do not always occur, and as such, protection is necessary.

Using the results from our study as a guide, we develop ARC (Automated Resiliency for Compression). ARC is a tool that automatically chooses and applies the optimal ECC configuration given user constraints on storage, throughput, and resiliency. When designing ARC, we set four main goals: scalability, performance, resiliency, and ease of use. When evaluating ARC, we begin by evaluating the scalability of its four underlying ECC algorithms. From this evaluation, we find each ECC algorithm scales near linearly with encoding throughputs ranging from 0.04 – 3730 MB/s and decoding throughputs ranging from 10.64 – 3602 MB/s when using a 40 core node. Next, we find ARC satisfies user constraints whether they synergize well or conflict with one another. We also find that ARC handles both single-bit and multi-bit soft errors effectively depending on the constraints the user provides. Finally, we illustrate that it only takes four lines of code to implement ARC and demonstrate how users should consider a systems failure rate to choose the most optimal constraints for the system they are running on.

In conclusion, the work in this thesis contributes a better understanding of how soft errors impact data compressed using two leading lossy compression algorithms, SZ and ZFP. From our findings, it is clear that data compressed using either algorithm is susceptible to soft errors and requires extra protection to ensure data fidelity. As a result, we develop ARC to effectively and efficiently protect lossy compressed data while abiding by user constraints on storage, throughput, and resiliency. Our evaluation of ARC indicates that it is a powerful tool that is more than capable

of assisting users in protecting their data.

In the future, we aim to further improve ARC's abilities in various ways. First, we aim to add more ECC algorithms and improve existing ECC implementations such that ARC is capable of achieving even higher throughputs. Next, we plan to utilize other parallelization paradigms like MPI or GPUs to further increase ARC's achievable throughput. Finally, we aim to design a streamlined API to simplify the addition of custom ECC algorithms and user constraints.

Bibliography

[1] Hurricane isabel simulation data. [Online]. This data was produced by the Weather Research and Forecast model, courtesy of NCAR, and the U.S. National Science Foundation. Available at http://vis.computer.org/vis2004contest/data.html.

[2] Ann S. Almgren, John B. Bell, Mike J. Lijewski, Zarija Lukić, and Ethan Van Andel. Nyx: A MASSIVELY PARALLEL AMR CODE FOR COMPUTATIONAL COSMOLOGY. *The Astrophysical Journal*, 765(1):39, feb 2013.

[3] Serhiy Avramenko, Matteo Sonza Reorda, Massimo Violante, and Görschwin Fey. A high-level approach to analyze the effects of soft errors on lossless compression algorithms. *Journal of Electronic Testing*, 33(1):53–64, 2017.

[4] A. H. Baker, D. Hammerling, and H. Xu. Optimizing data compression for the community earth system model. *AGU Fall Meeting Abstracts*, 11, Dec 2019.

[5] Allison H. Baker, Dorit M. Hammerling, Sheri A. Mickelson, Haiying Xu, Martin B. Stolpe, Phillipe Naveau, Ben Sanderson, Imme Ebert-Uphoff, Savini Samarasinghe, Francesco De Simone, and et al. Evaluating lossy data compression on climate simulation data within a large ensemble. *Geoscientific Model Development*, 9(12):4381–4403, Dec 2016.

[6] Allison H. Baker, Haiying Xu, Dorit M. Hammerling, Shaomeng Li, and John P. Clyne. Toward a multi-method approach: Lossy data compression for climate simulation data. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing*, Lecture Notes in Computer Science, page 30–42. Springer International Publishing, 2017.

[7] Tommaso Benacchio, Luca Bonaventura, Mirco Altenbernd, Chris D Cantwell, Peter D Düben, Mike Gillard, Luc Giraud, Dominik Göddeke, Erwan Raffin, Keita Teranishi, et al. Resilience and fault-tolerance in high-performance computing for numerical weather and climate prediction.

[8] Jon Calhoun, Franck Cappello, Luke N Olson, Marc Snir, and William D Gropp. Exploring the feasibility of lossy compression for pde simulations. *The International Journal of High Performance Computing Applications*, 33(2):397–410, 2019.

[9] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications*, 33(6):1201–1220, Nov 2019.

[10] Yann Collet and Murray Kucherawy. Zstandard Compression and the application/zstd Media Type. RFC 8478, October 2018.

[11] Khanh N Dang, Michael Meyer, Yuichi Okuyama, and Abderazek Ben Abdallah. Reliability assessment and quantitative evaluation of soft-error resilient 3d network-on-chip systems. In *2016 IEEE 25th Asian Test Symposium (ATS)*, pages 161–166. IEEE, 2016.

[12] S. Di and F. Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 730–739, May 2016.

[13] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 610–621. IEEE, 2014.

[14] Bo Fang. *Approaches for building error resilient applications*. PhD thesis, University of British Columbia, 2020.

[15] J. L. Gailly. Gzip, 1992.

[16] A. M. Gok, S. Di, Y. Alexeev, D. Tao, V. Mironov, X. Liang, and F. Cappello. Pastri: Error-bounded lossy compression for two-electron integrals in quantum chemistry. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, page 1–11, Sep 2018.

[17] Thaylon Guedes, Leonardo A Jesus, Kary ACS Ocaña, Lucia MA Drummond, and Daniel de Oliveira. Provenance-based fault tolerance technique recommendation for cloud-based scientific workflows: a practical approach. *Cluster Computing*, 23(1):123–148, 2020.

[18] James W. Hurrell, M. M. Holland, P. R. Gent, S. Ghan, Jennifer E. Kay, P. J. Kushner, J.-F. Lamarque, W. G. Large, D. Lawrence, K. Lindsay, and et al. The community earth system model: A framework for collaborative research. *Bulletin of the American Meteorological Society*, 94(9):1339–1360, Sep 2013.

[19] Andy A Hwang, Ioan A Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design. *ACM SIGPLAN Notices*, 47(4):111–122, 2012.

[20] D. Laney, S. Langer, C. Weber, P. Lindstrom, and A. Wegener. Assessing the effects of data compression in simulations using physically motivated metrics. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 1–12, Nov 2013.

[21] Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. Sdc resilient error-bounded lossy compressor, 2020.

[22] Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. Towards end-to-end sdc detection for hpc applications equipped with lossy compression. In *Proceedings of the 22nd IEEE International Conference on Cluster Computing*. IEEE, 2020.

[23] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, page 438–447, Dec 2018.

[24] Xin Liang, Ben Whitney, Jieyang Chen, Lipeng Wan, Qing Liu, Dingwen Tao, James Kress, Dave Pugmire, Matthew Wolf, Norbert Podhorszki, and et al. Mgard+: Optimizing multilevel methods for error-bounded scientific data reduction. *arXiv:2010.05872 [cs]*, Nov 2020. arXiv: 2010.05872.

[25] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, Dec 2014.

[26] Joseph Nardi, Noah Feldman, Andrew Poppick, Allison Baker, and Dorit Hammerling. Statistical analysis of compressed climate data. Technical report, NCAR, 2018.

[27] C. Nguyen and G. R. Redinbo. Fault tolerance design in jpeg 2000 image compression system. *IEEE Transactions on Dependable and Secure Computing*, 2(1):57–75, 2005.

[28] Xiang Ni, Tanzima Islam, Kathryn Mohror, Adam Moody, and Laxmikant V Kale. Lossy compression for checkpointing: Fallible or feasible? In *Poster Session of the 2014 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, Washington, DC, USA, 2014. IEEE Computer Society.

[29] Mizanur Rahman, Mhafuzul Islam, Jon Calhoun, and Mashrur Chowdhury. Real-time pedestrian detection approach with an efficient data communication bandwidth strategy. *Transportation Research Record*, 2673(6):129–139, Jun 2019.

[30] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. Exploration of lossy compression for application-level checkpoint/restart. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, pages 914–922, Washington, DC, USA, 2015. IEEE Computer Society.

[31] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. *Communications of the ACM*, 54(2):100–107, 2011.

[32] Baodi Shan, Aabid Shamji, Jiannan Tian, Guanpeng Li, and Dingwen Tao. Lcfi: A fault injection tool for studying lossy compression error propagation in hpc programs, 2020.

[33] Taniya Siddiqua, Athanasios E Papathanasiou, Arijit Biswas, and Sudhanva Gurumurthi. Analysis and modeling of memory errors from large-scale field data collection. In *SELSE*, 2013.

[34] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei keng Liao, and Alok Choudhary. Data compression for the exascale computing era - survey. *Supercomputing frontiers and innovations*, 1(2), 2014.

[35] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *SIGARCH Comput. Archit. News*, 43(1):297–310, March 2015.

[36] Vilas Sridharan and Dean Liberty. A study of dram failures in the field. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[37] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory positional effects in dram and sram faults. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2013.

[38] Li Tan and Nathan DeBardeleben. Failure analysis and quantification for contemporary and future supercomputers. *arXiv preprint arXiv:1911.02118*, 2019.

[39] D. Tao, S. Di, Z. Chen, and F. Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 1129–1139, May 2017.

[40] P. Triantafyllides, T. Reza, and J. C. Calhoun. Analyzing the impact of lossy compressor variability on checkpointing scientific simulations. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, page 1–5, Sep 2019.

[41] R. Underwood. Libpressio, 2020.

[42] X. Wei, R. Zhang, Y. Liu, H. Yue, and J. Tan. Evaluating the soft error resilience of instructions for gpu applications. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 459–464, 2019.

[43] Xin-Chuan Wu, Sheng Di, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T. Chong. Memory-efficient quantum circuit simulation by using lossy data compression. Nov 2018.